PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 17/30, 17/60

A1

(11) International Publication Number: **WO 99/59083**

(43) International Publication Date: 18 November 1999 (18.11.99)

(21) International Application Number: PCT/IB99/00878

(22) International Filing Date: 14 May 1999 (14.05.99)

(30) Priority Data:

9810368.2	14 May 1998 (14.05.98)	GB
9812183.3	5 June 1998 (05.06.98)	GB
9909599.4	26 April 1999 (26.04.99)	GB

(71) Applicant *(for all designated States except US)*: PEOPLEDOC LTD. [GB/GB]; 14 Links Place, Leith, Edinburgh EH6 7EZ (GB).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: WRIGHT, Graham [GB/GB]; 12 Merchiston Grove, Edinburgh EH11 1PW (GB). GILLESPIE, Alan [GB/GB]; 5/39 Mid Steil, Glenlockhart, Edinburgh EH10 5XB (GB).

(74) Agent: PLOUGMANN, VINGTOFT & PARTNERS A/S; Sankt Annæ Plads 11, P.O. Box 3007, DK–1021 Copenhagen K (DK).

(81) Designated States: AE, AL, AM, AT, AT (Utility model), AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, CZ (Utility model), DE, DE (Utility model), DK, DK (Utility model), EE, EE (Utility model), ES, FI, FI (Utility model), GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SK (Utility model), SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

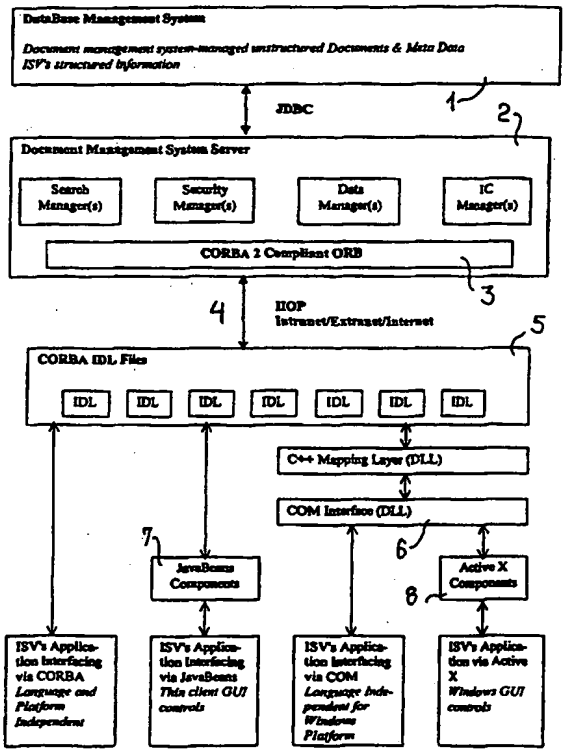
Published

With international search report.

(54) Title: A DOCUMENT STORING AND RETRIEVING SYSTEM AND A SOFTWARE APPLICATION SYSTEM INTEGRATING A DOCUMENT STORING AND RETRIEVING SYSTEM

(57) Abstract

A document storing and retrieving system in a networking environment comprising a plurality of individual components, wherein the components interact and may be interfaced with external systems through platform independent communication mechanisms, i.e. JAVA, CORBA. The system may store and retrieve meta data (index data) as well as documents into and from the same database so that both documents and meta data pertaining to the respective documents may be contained within the same database. Preferably, the document management system communicates with the database through a standard, platform–independent communication mechanism, such as JDBC. In particular, the invention concerns a system comprising at least one system controlled scanner for scanning paper documents and producing graphic image files representing the paper documents.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

A DOCUMENT STORING AND RETRIEVING SYSTEM AND A SOFTWARE APPLICATION SYSTEM INTEGRATING A DOCUMENT STORING AND RETRIEVING SYSTEM

Brief disclosure of the invention

The present invention concerns software application systems that integrate, or , expressed in another manner, are integrated with a document management system for storing and retrieving documents, the document managing system comprising a plurality of individual components which interact and interface with and may be controlled by the software application system and optionally other external systems through platform-independent communication mechanisms. The invention also relates to a document management system having integratability, preferably a multiple level integratability, with systems that are external to the document management system.

The document management system may store and retrieve meta data (index data) as well as documents into and from the same database so that both documents and meta data pertaining to the respective documents may be contained within the same database. Preferably, the document management system communicates with the database through a standard, platform-independent communication mechanism, such as JDBC.

Background

The known document management systems widely used at present are highly specialised systems which are optimised from the point of view of functionality and features.

US 5,628,003 discloses a system for storage and retrieval of documents, wherein a paper document is scanned, the image file is processed by an optical character recognition module and the resulting text file is stored together with the image file. Thus, text search may be performed by a user in the text file and the user is able to view the search results as images of the full documents, including mathematical expressions etc.

A number of existing document management systems have been and are being updated in order to be in accordance with the steadily more widespread use of and increasingly importance of network-based computer environments, the application are being so-called "Web-enabled".

NovaSoft Systems, Inc. described in 1997 in an article "Process-centric Application Drive the Bottom Line: NovaSoft Marries the Web and the Business Process" published on their homepage (http:\\www.novasoft.com\web \press\aberdeen.html) how the document management system have been modified and will be modified in the future to be able to support a vast number of the Internet standards, such as Java, Active X, HTTP, RMI, IIOP, DCOM etc. so that the system is enabled to communicate and perform information management via the Internet with other Web-enabled applications.

Grala et al. disclosed in an article: "GDOC: a system for storage and authoring of documents through WEB browsers" in proceedings form 17th International Conference of the Chilean Computer Science Society, Valparaiso, Chile 10-15 November 1997 (pages 115-124) a document management system having components based on open standard, thus leading to a platform-independent software. The system stores the data contents, such as the text, of a document and the data defining the presentation of the data, the logical structure, separately but within the same database. The users communicate with the system via an HTML-browser using the HTTP Transfer Protocol.

Most office software packages do not comprise full-fledged document management systems, for which reason a document management system is normally acquired as a separate system to be connected with the office existing applications such as word processing software, e-mail software, etc. Thus, document management systems which can be connected with existing applications, such as Windows-compatible office applications, on an output level are known.

With the increasing importance of work group applications, intranet usage, Internet usage and telecommunication and with the increasing importance in most organisations of one or more software application systems being line of business

systems by use of which a significant amount of the production of the organisation is performed, the monolithic character of conventional document management systems is a severe restriction and complication limiting their integration in a modern internal and internal/external business system such as an office environment.

A common problem related to computer-networking and sharing of information within an organisation is that the computers operate on different platforms, depending on the type of computer, such as mainframes, personal computers, file servers, etc., for which reason the users have difficulties with operating applications residing on a computer using one platform remotely from a computer using a second platform. Another common problem is that an application which is to be used on a number of different platforms within an organisation has to be tailor-made for each platform. Yet another common problem is that applications residing on a number of computers within the organisation must be updated on each of these computers, preferably simultaneously, when a new version of the application is introduced.

An increasing demand exists for integrating management of conventional electronic documents such as word processing files and spreadsheet files with management of data files representing physical phenomena, such as audio files, video files and image files, in particular image files representing paper documents, in such a manner that the various file types are managed in the same system. The devices from which the data files representing physical phenomena are obtained should be controlled from the data management system so that the files are obtained, indexed and stored in a database via the same user interface in order to make the system efficient in use. In particular, the need for integrating large amounts of scanned paper documents into document management systems has created a demand for flexible, general purpose document management systems wherein the scanning procedure, indexing procedure and storing are integrated.

Thus, it would be highly desirable to provide a document management system which could be more easily integrated in and controlled from software application systems environments and architecture and be independent of system platforms, both with respect to computer operating systems and with respect to platforms defined by specific software producers. It would also be highly desirable that the documents

management system and the software application system may be operated from the same platform and/or that they may be use the same database for managing structured data from the application system as well as unstructured data from the document management system.

Description of the invention

The present invention relates to software application systems being integrated with a document management system which, in contrast to the monolithic integratability of the known systems, provides a multiple level integratability with other systems, and which, in addition, provides a high degree of freedom with respect to the architecture of the document management system itself. Indeed, in preferred embodiments, the document management system of the present invention could, because of its multi-level integrating functions, be the system that exerts the integrating function in a software application environment comprising line of business application systems as well as other office application systems.

It is a key issue of the present invention that the document management system according to the invention is platform-independent in order to provide the multiple level integratability with other systems and enable the system to be implemented on various platforms and to interact with systems being implemented on the same or various other platforms. An important feature of the system according to the invention is that the document management system in contrast to most known systems may store and retrieve meta data (index data, data describing documents) as well as documents proper into and from, respectively, the same database so that both documents and meta data pertaining to the respective documents may be contained within the same database. The meta data for at least some of the documents managed by the document management system may even come from the software application system directly. In this way, the document management system is only required to be enabled to communicate with one database (or database system) in order to manage the documents, contrary to most known document management systems, in which the meta data are stored in a database together with information about the location of the document data themselves, the document data themselves being stored in one or more other databases or file systems. However, this/these

databases or file system(s) may be situated on one or more of various platforms each having their own type of file system and the document management system must thus be enabled to communicate with a variety of file systems in order to able to operate with such systems. Preferably, the document management system of the present invention communicates with the database through a standard, platform-independent communication mechanism, such as JDBC.

The document management system according to the present invention may be implemented as a server-server solution in which the software application system server and the document management system server communicates and the software application system or components thereof may control the operation of the document management system. The document management system is preferably furthermore implemented as a client-server solution, where the components of the document management system reside on a limited number of servers which may be accessed by a number of client that uses a general-purpose browser for establishing a session on the system. This arrangement enables simple updating of software when new versions are available and makes it possible for users to access the system from any computer connected to the same network as a server, without first having to install the software on the computer. The client runs as a so-called thin client, and the system is very suitable for using so-called net computers as clients.

The document management system of the present invention should in preferred embodiments be able to handle both paper documents which are scanned so as to produce a graphic image data file, that represents the content of the paper document and may be handled and stored like any other document, electronic documents, that have been received or retrieved from other systems of the same or other types, connected in a local and/or a public network, and electronic documents, that have been provided by the users of the system. Optionally, the system may also be able to handle electronic documents, that reside in databases or file storage systems, that are not part of the document management system, where only information about the document and the location of the document is stored in a database within the document management system, but not the electronic document itself. Such a system would be able to handle nearly any kind of document that occurs within an organisation so that the documents may be stored in and retrieved from a database by

the same system and the users have access to all documents in the organisation from one user-interface, preferably a graphical user-interface.

Thus, it is an object of the present invention to provide an integration of a software application system and a document management system through platform-independent communication mechanisms.

It is a further object of the present invention to provide a system for document management having a high degree of platform-independency and uses common standard application program interface communication mechanisms so that it may integrate with a vast number of systems. Thus, it is an object of the present invention to provide a system for document management that may be operated from a number of different platforms or operating systems, such as versions of Windows for Personal Computers, Windows NT, different versions and varieties of UNIX, OS/2, AS/400, etc. The system should be enabled to operate from these platforms simultaneously, so that the system may reside on a computer using one platform and be operated from another computer using a second platform via a network. Also, the system should not be dependent on platforms defined by a specific software producers, but rather be designed according to common standards and platforms and the components of the system should be able to interface according to common standards and platforms.

It is a still further object of the invention to provide a system that is constructed from a number of individual components that may be operated individually, and where these components interact with each other through platform-independent mechanisms, so that the individual components residing on a computer using one platform may interact with other individual components residing on another computer using a second platform via a network.

It is a yet further object of the invention to provide a system where the components of the document management system resides on one or more servers and are accessible from several computers in a network, clients, so that the software is downloaded to the clients when a session on the system is established by a client. Thereby, an update to a new software version needs only to be performed on a limited number of servers.

It is a yet still further object of the invention to provide a system where the individual components may interact with external systems through platform-independent mechanisms, such as CORBA and COM, so that the individual components may be accessed from external systems, thus enabling a far more versatile integration on multiple levels with other systems instead of interaction on input/output level of the whole system. Such external systems may be work flow applications, accounting systems, word processors, Internet/Intranet applications, such as browsers, search engines etc., mail systems, spreadsheets, systems that integrates several applications, so-called office packages, etc.

It is a still further object of the invention to provide a system, wherein one or more scanners for scanning paper documents are controlled by the system so that the whole process of scanning a paper document and obtaining a graphic image data file, creating an index card and optionally other identification and characterisation of the paper document and storage of the data in a database is performed within the same system.

In one aspect, the invention concerns a software application system for use in a computer network, the network normally being selected from the group consisting of a local area network (LAN), a wide area network (WAN), a public network and any combination thereof, the software application system comprising

a database with which the software application system may communicate so as to store structured data in and retrieve structured data from said database, and,

integrated with the software application system, a document management system comprising one or more individual components capable of interfacing with the software application system through platform-independent communication mechanisms, the interfacing enabling the software application system to control storage of documents and meta data related to the documents and retrieval of documents and meta data by the document management system,

the components of the document management system residing, or being capable of residing, on one or several computers, and being accessible from one or more computers connected to the network, and the document management system further comprising

a database with which one or more of said individual components of the document management system may communicate so as to manage documents and meta data relating to the documents, including storing documents and meta data and retrieving documents and meta data from said database, the database containing both documents and meta data pertaining to the respective documents.

The network that interconnects the computers may be any kind of network or combination of networks, that enables data communication between the computers, preferably via the TCP/IP protocol. The individual computer may be connected to the network when it is desired to establish a session on the system or another system on the networked computers and otherwise run as a stand-alone unit, or the individual computer may be permanently connected to the network.

The individual components may be any component that is adapted for performing the required operations, such as computer software, an integrated circuit, a programmable integrated circuit, etc.

The database of the document management system is preferably the same database as is employed by the software application system so that structured data from the software application system and structured as well as unstructured data (documents) from the document management system may be managed within the same database, thus providing access, within the same database, to the structured data of the software application system, the meta data pertaining to the documents, and the documents proper, thereby ensuring the highest degree of flexibility with respect to setting up routines, views, queries and functions in the software application system which access or utilise the documents in connection with the software application system, including such routines, views, queries or functions which access documents which are relevant in connection with a particular condition or set of conditions or a particular view of query in the software application system.

The preferred database type for the document management system database (which, thus, preferably is also the the database used by the software application system) is a relational or an object database which advantageously is accessible through a standard communication mechanism so that the interfacing between the document management system database and one or more of the individual components of the document management system may be through a platform-independent communication mechanism such as JDBC or a variant or derivative thereof.

The preferred database for the document management system database is a relational or an object database which advantageously is accessible through a standard communication mechanism so that the interfacing between the document management system database and one or more of the individual components of the document management system may be through a platform-independent communication mechanism such as JDBC or a variant or derivative thereof.

At least some and preferably all of the mentioned individual components of the document management system are implemented in a platform-independent programming language such as Java or a variant or derivative thereof so as to obtain a high level of platform-independency of the system.

In particular, the individual components of the software application system implemented in the platform-independent programming language may advantageously be organised according to standards for creating components in that language such as Java Beans or a variant or derivative thereof.

The platform-independent communication mechanisms mentioned previously, through which the software application system, or one or several components thereof, is interfacing with the document management system or components thereof is preferably CORBA or a variant or derivative thereof or alternatively COM. Likewise, the interaction between at least some of said individual components of the software application system as well as the interaction between individual components of the document management is preferably also through platform-independent communication mechanisms such as CORBA or a variant or derivative thereof.

Preferably, the document management system or one or more individual components thereof is/are capable of interfacing with external systems or components thereof, including the software application system or components thereof through platform-independent communication mechanisms, such as CORBA or a variant or derivative thereof and/or alternatively COM or a variant or derivative thereof.

The document management system may also be enabled to establish a client-server interfacing. Thus, individual components of the document management system residing on individual computers connected to the network may enable said computers to interface with the document management system or components thereof through platform-independent communication mechanism, such as CORBA or a variant or derivative thereof, the interfacing enabling the individual computer to control storage of documents and meta data and retrieval of documents and meta data by the document management system. The individual components may be accessed via the network and optionally downloaded onto a computer connected to the network, preferably upon establishment of connection via the network between the document management system and the computer so that the computer is a so-called "thin" client that does not require that any component of the system resides on the computer before a connection to the system is established. This also means that updating of the system only requires that the components residing permanently on a number of the networked computers, the servers, are updated. Such a connection between the document management system and a computer connected to the network may established with a general purpose interfacing software, such as a browser, operated by said computer. The browser is only used to establish the contact between the system and the computer and for downloading of the components. The interfacing itself via the network between the document management system and the computer in which documents as well as meta data may be exchanged via the interfacing and the operation of the document management system may be controlled from the computer is performed using the Internet Inter-orb Protocol (IIOP) or a variant or derivative thereof. Data are transferred much faster using this protocol, as much as 200 times faster, compared to the use of http which is normally employed by general purpose browsers. The browser may be of any type that can use a standard compatible with a standard of the system. Today, the most

popular browsers are Netscape Navigator and Microsoft Internet Explorer. The individual components are preferably implemented in a platform-independent programming language such as Java or a variant or derivative thereof and they may also be organised according to standards for creating components in that language such as Java Beans or a variant or derivative thereof.

The software application system as well as other optional systems and computers interfaces preferably with the document management system via one or more layers selected from the group consisting of CORBA IDL files, Component Object Model interfaces, Java Beans components and Active X components or variants or derivatives thereof and the software application system as well as other optional systems and computers interfacing with the document management system may comprise individual component, preferably implemented in Java or a variant or derivative thereof, interfacing the one or more layers and the document management system by means of platform-independent communication mechanism, such as CORBA or a variant or derivative thereof. See the description related to Fig. 1 below for detail about a preferred embodiment of the interfacing of the system through one or more of the above-mentioned layers.

The document management system according to the invention may in a preferred embodiment comprise at least the following components:

- an index card component for creating index cards for new documents to be stored by the system and for editing existing index cards, the index cards comprising meta data pertaining the respective documents,
- an archive component for creating and updating documents in the database, each document comprising at least one index card relating to the document data and optionally one or more sets of data, such as graphic image data or text data, and
- a search/retrieve component for searching the database for documents according to a set of search criteria and producing a set of data representing the result of the search, and for retrieving documents from the database.

Furthermore, the document management system may additionally comprise one or several of the following components:

- a printing component for printing sets of data, such as graphic image data or text data, that have been received by the system or that have been retrieved from the database,
- a security component for imposing regulations on the accessibility of documents in the database for the users of the system,
- a viewing component for producing a representation on a graphical display of sets of data, such as graphic image data or text data, that have been received by the system from external systems or components, or that have been retrieved from the database, and
- a directory component for producing a representation on a graphical display of the content of a file storage means, allowing a user to select a file from the file storage means and adding said file to a document in the database.

The document management system may in a further preferred embodiment comprise

- at least one peripheral means capable of producing an output comprising data representing a physical phenomenon external to the system (which output then becomes input to the document management system), the at least one peripheral means and at least one of said computers being interconnected, optionally on the network, and the control of the operation of the at least one peripheral means, including handling of output and storage of at least a part of the output as at least a part of a document in the database, being performed by one or more of said individual components.

The at least one peripheral means capable of producing an output comprising data representing a physical phenomenon external to the system may be a device for recording sounds comprising a microphone, a digital single-frame camera, a video camera, etc. However, in a preferred embodiment of the invention, at least one of the at least one peripheral means is a device generating an output representing digitized graphical image data, such as a scanner for scanning paper documents and producing an output file comprising graphic image data representing the paper document. The system according to the invention may comprise one or more of any type of the peripheral means or a combination of one or more or any type of the peripheral means.

The interfacing between the at least one peripheral means and the components that control the operation of the peripheral means is preferably through platform-independent communication mechanisms. In particular, for embodiments of the system, wherein at least one of the at least one peripheral means is a device generating an output representing digitized graphical image data, such as a scanner, the platform-independent communication mechanisms through which the at least one device and the components that control the operation of the scanner is preferably TWAIN or a variant or derivative thereof. Alternatively, the platform-independent communication mechanisms through which at least one scanner and the components that control the operation of the scanner may be ISIS or a modification or a variant or derivative thereof.

One or more of the peripheral means may alternatively or additionally be capable of producing an output comprising data representing sound and/or be capable of producing an output comprising data representing images such as digital cameras.

The document management system should preferably in any case comprise a controlling component for controlling the operation of the peripheral means and handling the output from the peripheral means.

In a particular embodiment of the system according to the invention, it comprises an optical character recognition module for analysing the content of a set of graphic image data representing a document containing textual matter and producing an output file comprising text data, that represent at least a part of said textual matter, the control of the operation of the optical character recognition module, including handling of output, being performed by one or more of said individual components.

The textual matter that is found by the optical character recognition module may be information regarding the indexing of the document, such as reference code(s), attention person, date, senders identity, etc., or it may be a full-text conversion of a written document.

At least one of the computers connected to the network and interacting with the software application system and/or the document management system preferably comprises a graphical display and input means, such as a keyboard and optionally a pointing device, such as a mouse, the communication of information from the system to a user of the system being performed with the use of a graphical user interface displayed on the graphical display, and the communication of information from the user to the system being performed with the input means.

The term "software application system" is understood in a broad sense but the systems to be used with the present invention are preferably so-called line of business application system, such as a manufacturing system, an accounting system, a Custom Relationship Management (CRM) system, an Enterprise Resource Planning (ERP) system, a health care system, a financial application, and other software systems by use of which an important part of an organisations production is performed.

The present invention further relates to a document management system suitable for use in establishing a software application system with integrated document management system as described above, the document management system comprising:

one or more individual components residing on one or several computers, and being accessible from one or more computers connected to the network, and

a database with which one or more of said individual components of the document management system may communicate so as to manage documents and meta data pertaining to the documents, including storing documents with meta data relating to the documents in the database and retrieving documents and meta data from the database, the database containing both documents and meta data pertaining to the respective documents, and

integration layers enabling integration between the document management system and the software application system in such a manner that the document management system can be controlled from within the software application.

The interaction layers in such a document management system are preferably selected from the group consisting of CORBA IDL files, COM interfaces, Java Beans components and Active X components or variants or derivatives thereof and the document management system may advantageously be implemented in Java or a variant or derivative thereof.

The document management system according to the invention may also have the features for the document management system previously.

Brief description of figures

Most of the figures are class diagram and sequence diagrams that comply with the common standard for such diagrams, the Unified Modelling Language (UML), developed at Rational Software Corporation.

Fig. 1 shows an architectural model of the system,

Fig. 2 shows another architectural model of the interfacing of the document management system and a client,

Fig. 3 shows the vertical architecture of the system,

Fig. 4 shows a hierarchy as presented by the "tree view",

Fig. 5 represents the interaction between the Archive Manager and other system components,

Fig. 6 models the relationships between the Archive Manager and the other components and classes concerned with the storage process,

Fig. 7 is a sequence diagram showing the messages and events that occur when a user chooses to create a new document and commit it to the system,

Fig. 8 is a class diagram showing the relationships between the Retrieve Manager and the other components and classes concerned with the retrieve process,

Fig. 9 is a sequence diagram that outlines the messages and events generated for each type of search that can be carried out during the retrieval process,

Fig. 10 is a sequence diagram representing a free text search,

Fig. 11 is a sequence diagram for browse searching,

Fig. 12 is a sequence diagram representing the initial stages of a browse search,

Fig. 13 shows the flow of control within the beans of the Index Card Component,

Fig. 14 is a sequence diagram modelling "Creating an Index Card Template and adding a new List Field",

Fig. 15 is a sequence diagram modelling "Creating an Index Card Template, adding a new List Field and selecting a predefined List Items",

Fig. 16 is a sequence diagram modelling "Creating an Index Card Template and adding a new Text Field",

Fig. 17 is a class diagram modelling "Creating an Index Card Template and adding a new List Field",

Fig. 18 is a sequence diagram modelling "Wiring the Create Index Card JavaBeans",

Fig. 19 is a sequence diagram modelling "Wiring the Controllers to their GUIs",

Figs. 20 and 21 are class diagrams modelling "Wiring the Create Index Card Template JavaBeans",

Figs. 22 and 23 are sequence diagrams modelling "Commit Index Card" and "Populate Index Card",

Fig. 24 is a class diagram modelling "Commit and Populate Index Card",

Fig. 25 is a sequence diagram modelling "Index the document",

Figs. 26 and 27 are sequences diagrams modelling "View the Index Card",

Fig. 28 is a class diagram modelling the "Create List Items bean",

Fig. 29 shows the flow of control within the Directory Data Source Component,

Fig. 30 is a class diagram modelling the Directory Data Source Component,

Fig. 31 is a sequence diagram showing the construction of the objects in the Directory Data Source Component and how they are wired up to listen for events.

Fig. 32 is a sequence diagram showing the interaction between objects Directory Data Source Component as the user parses the directory tree to select a file to add to the archive,

Fig. 33 presents the various parts of the Scanner component and their interaction among each other and with some other system components,

Fig. 34 is a class diagram that models in greater detail the inner structure of the Scanner component,

Fig. 35 is a class diagram modelling the inner structure of the CSAdapter and its relation to the Server and Controller class of the Scanner component,

Fig. 36 is a class diagram that models the typical inner structure of the USAdapter and its relation to the Server and User class of the Scanner component,

Fig. 37 shows the data classes JobData and ScannerSpec,

Fig. 38 is a sequence diagram modelling the sequence "Scan Document",

Fig. 39 is a chain sequence that models adding or re-scanning of a single image,

Fig. 40 is a class diagram that shows the types of objects present in the Audit bean and the relationships between them,

Fig. 41 is a sequence diagram exploring the steps that are taken when an AuditEvent is generated and fired at an AuditEventListener,

Fig. 42 shows the basic architecture of the Security Manager component,

Fig. 43 is a class diagram showing the main types of objects present in the Security Management Component and the relationships between them,

Fig. 44 is a sequence diagram exploring the events that occur when security is added to a document,

Fig. 45 is a sequence diagram exploring the events that occur when the security is modified on a document,

Fig. 46 is a sequence diagram exploring the events that occur when a new user is created on the system,

Fig. 47 is a sequence diagram exploring the events that occur when the details of a user are modified,

Fig. 48 is a sequence diagram exploring the events that occur when a new group is added to the system,

Fig. 49 shows the basic architecture of the Resource Management component,

Fig. 50 is a class diagram showing the types of objects present in Resource Management and the relationships between them, and

Fig. 51 is a sequence diagram exploring the steps that are taken when a GUI requests a string from a Local Resource Manager.

Preferred embodiment of the system.

This section provides a high-level description of a preferred embodiment of the system according to the invention. The main components and there responsibilities are outlined in this section and a more detailed description of the components is given subsequently.

The main purpose of the system is storage and retrieval of electronic documents across a distributed environment wherein the document management system is integrated with and controllable from a software application system.

The stored documents can be made up of files of any data format such as Microsoft Office documents, scanned image files, audio files, ASCII text files etc. Input of these files into the system should be as simple as possible, integrating with external systems wherever possible.

Associated with the documents are one or more structured index cards that will be user-configurable and contain information to categorize the documents. This information could include such things as company names, product information, invoice dates, contact names etc.

Retrieval of the documents will be based on either this structured index information or the content of the document, e.g. a search could be performed to find any documents containing the word 'Internet'.

Security restrictions can be applied to control access rights to any document. This ensures that only specified users can view and/or edit these secured documents.

The document management system comprises a set of components that are 'wired' together in order to achieve the overall system functionality. The persistent data of the system consists of the classes in the domain package. All the components in the system act on these domain classes to achieve their functionality. The domain classes include the structure of a document with its index cards, data, security and its unique object identifier (OID).

The main document management system components are

- Archive Component - responsible for the creation and storage of documents.
- Retrieve Component - responsible for retrieving documents
- Index Card Manager Component - responsible for the creation and management of index cards.
- Directory Component - responsible for providing files from a file system.
- Scan Component - responsible for providing scanned image files.
- In-Tray Component - responsible for providing external file input.
- Free Text Manager Component - responsible for ensuring that documents are free text stored and retrieved.
- Annotation Component - responsible for displaying annotations
- Audit Component - responsible for recording audible actions that happen within the system.
- Security Component - responsible for managing users and creating and managing document security.
- Document View Component - responsible for displaying document data.
- Log In Component - initial application interface.
- Internationalisation/Resource Management.

These components are completely self contained, only being dependant on the domain classes. The system is capable of having any one of these components replaced by a component that can perform the same task with no impact on any other components.

The domain package and components of the document management system are described in more detail below.

Domain Model

The domain models the data central to the system. All components within the system will act on this model in order to create or manage any part of the system. All objects within the model can be made persistent.

The document is the placeholder for all constituent objects that make up a system document. These objects are a data container, document security, one or more index cards and a universal identifier. Communication of documents through the system is achieved by passing references to document objects.

The data container is the container for one or more data files. It provides methods for managing and navigating the files it contains. The data file is the storage object for binary data from files. An annotation contains the information required to render an annotation, e.g. for text annotations this may include the text, font size and colour and positional information etc.

Document security contains a list of all users and user groups that have access rights to the document. The document is considered to be unsecured if there are no references to users or user groups. When a user tries to retrieve a document, access will be provided if the document is unsecured or if the user is named specifically in the document security or the user is in a user group that is named in the document security.

A user identifies a person that can use the system. A user will have a user name and an associated password that are required to log in to the system. The user will also have system permissions that define specific rights for the user, e.g. permission to create documents or create index cards, etc. A user group is a collection of users.

An index card is a collection of structured index information that can be used to store and later retrieve a document. It is made up of index card data and an index card template. The index card data contains all the field templates that are on the index card and their corresponding data. The index card template contains positional information about the appearance of the index card fields and the specification of

which data the associated field data can hold. The index card field data stores the index data for the associated index card field template.

The document unique object identifier (OID) will uniquely identify a document across multiple systems of the present type and across multiple electronic storage systems that conform to some industry standard (e.g. ODMA). This will allow future interoperability between the present system and other storage systems.

Persistence

All classes within the domain model need to be stored persistently. This includes:

- Configuration data - System configuration, index card set-up, user data etc.
- Application data - data file storage, index card data, audit data etc.

The application layer of the system will commit objects to the system when it has responsibility to ensure that they are stored persistently. Each object that can be stored persistently will have a corresponding broker class and server class.

When an object needs to be persistent it will externalize its state to a data stream. The broker will know the format of the stream and can read this data and store it in a third party database. The communication between the broker and database is based on JDBC, an open database format which is supported for most main databases, e.g. Microsoft SQL, Oracle, Informix etc. This ensures that the system is not dependent on any particular database.

The reverse happens when an object needs to be recovered from the persistent store, i.e. the broker will retrieve the data from the database using JDBC and write it to a data stream. The object will then internalize its state using this data.

The above process ensures that the object is completely responsible for its own internal state and does not have to expose any private member variables to enable some third party to store its state.

Architectural model of the system

Fig. 1 is a schematic diagram showing an architectural model of a document management system according to the invention with its integration points or integration levels into an independent software application such as a line of business system. In a preferred embodiment, the document management system is a component-based native Internet document management system designed to be incorporated into independent software applications such as line of business applications.

By building the document management system into their software products, Independent Software Vendors (ISVs) can quickly offer differentiating document management features, within their own application, to their customer base. This is made possible through the platform-independence of the document management system and the language- and platform-independent interfacing of the system.

The system comprises a database 1 which is managed by a database management system and which may be the same relational or object database as is used by the line of business system or a separate relational or object database. The database is a database which is capable of storing the various types of files handled by the document management system, including both the structured information represented by the index data or meta data of the document management system and the "unstructured" information represented by the various types of documents, such as word processing documents, voice files, video files, image files, graphic vector files, etc., relevant to the line of business system and handled by the document management system. The database is an "open" database, that is, a database that can be accessed using a standard platform-independent communication mechanism such as JDBC (Java DataBase Connectivity). Thus, the document management system can interoperate with all mainstream databases such as Oracle, MS SQL Server, DB2, etc. The document management system suitably stores all of the documents (unstructured information) into the database as Binary Large Objects (BLOBs) and additionally stores all of the meta data associated with the unstructured data in the database as well. In this way, ISVs incorporating the document management system into their own client/server software applications can ensure, if

desired, that the document management system stores unstructured information (documents) in the same database that holds the structured information associated with their product. It will be understood, however, that the term database is to be understood in a broad sense and will comprise also the case where the database is constituted or represented by multiple (distributed) components as long as it is perceived by and interfaced with the system as one database in the manner described above.

The functionality or functionalities of the document management system normally reside on one or several computers (document management system servers) 2. The functionality of the system consists of a number of functions or "managers" which control various functional aspects. In the case illustrated, the basic functionality of the document management system is constituted by the data manager(s) which handle(s) the storage of the documents and meta data in the database, and the search manager(s) which handle(s) the searching and retrieval, normally *via* the data manager(s), the IC manager(s) which handle(s) the generation of index data (meta data) and the storage thereof in the database, normally *via* the data manager(s), the archive manager(s) (not shown) which handles the storage of documents from external systems, normally *via* the data manager, and optionally a number of additional components, such as security manager(s) which handle the access permissions to the system. Each connection to the server or servers causes a suite of managers to be spawned, thus ensuring that the system architecture can scale to handle multitudes of connections. This means that a plurality of instances of each manager can execute simultaneously.

The functionalities of the server or servers are preferably written entirely in Java. This ensures that the document management system can run on all mainstream deployment platforms such as NT, Solaris, UNIX, AS/400, etc. Preferably, the functionalities are specifically optimised for the Java 2 platform or further developments, variants or derivatives thereof.

A platform-independent communication mechanism, in the case illustrated a CORBA 2 compliant Object Request Broker (ORB), is embedded within the document management server and it is this ORB (e.g., Visibroker from Inprise, California, U.S.A.)

that underpins the CORBA based language and platform independent integration capabilities of the system.

Importantly, due to the document management system being a native Internet application, the Internet Inter-Orb Protocol (IIOP) 4 is used as the communications transport between the document management server and interfacing applications. IIOP allows for interfacing across an Intranet, Extranet or the Internet itself. The document management system is therefore fully Internet enabled and ideal for managing unstructured information within the context of Internet-aware software applications.

Integration Layers

As appears from Fig. 1, the document management system can offer a complete suite of integration possibilities to ISVs who wish to build the system into their own applications. The system illustrated uses a layered approach to its interfacing API set, each API layer essentially being derived from the native (in the case shown CORBA) integration capabilities of the system.

As mentioned above, the API set 5, 6, 7, 8 (a plurality of multi-tier APIs ranging from basic CORBA IDL files to the functionally richer API components such as Java Beans and Active X) communicates via the IIOP to the document management system server. ISVs are therefore ensured that their own interfacing applications, whether interfacing at the front-end, back-end or both, are able to efficiently access the document management system functionality over an Intranet, Extranet or the Internet.

The integration suite illustrated consists of the following layers, each of which offers a potential interface point to an ISV:

IDL Files

This layer consists of a number of Interface Definition Language (IDL) files for the platform-independent communication mechanism, in Fig. 1 illustrated as CORBA IDL files. Each file represents a functional subset of the document management system, thus allowing integration without the inclusion of redundant information at the interfacing end.

The IDL files, being essentially interface definitions for the platform-independent communication mechanism, can be used with any language and on any platform supporting the platform-independent communication mechanism, such as a CORBA 2.0 ORB. Therefore, it is possible for, e.g., a back-end COBOL application designed for the AS/400 environment to have the document management system functionality integrated via this IDL layer. By the same token it is also possible for, e.g., a front-end C++ application to integrate via this layer.

COM Interface DLL (incorporating C++ mapping layer DLL)

In Fig. 1, the COM interface layer consists of a Windows platform Dynamic Link Library (DLL) aligned to a C++ mapping layer DLL. The purpose of the mapping layer DLL is to map the document management system calls supplied in the COM DLL to the CORBA IDL calls in the CORBA IDL layer.

The COM interface can be used on the Windows platform with popular development languages such as Visual Basic and Visual C++, therefore ensuring that ISVs can integrate the document management system functionality within the framework of their own preferred development environments.

JavaBeans Components

The system illustrated in Fig. 1 offers a suite of JavaBeans client components at this layer. The JavaBeans components are built on top of the CORBA IDL layer. The JavaBeans offer ISVs, who are building thin client web based interfaces to their own applications, a rapid application development option for building-in document management system functionality.

The JavaBeans, which may be offered as a suite of autonomous components, also allow ISVs to include only the components that are necessary to their own use of the document management system.

ActiveX Components

For ISVs with Windows based client/server applications, the ActiveX Components layer of the system illustrated in Fig. 1 offers rapid application development of GUI based document management functionality. The ActiveX components themselves are built on the COM interface DLL layer which in turn maps to the CORBA IDL interface layer of the document management system.

ISVs using development environments such as Visual Basic or Visual C++ in their own development efforts can very quickly add document management capabilities to their applications, along with associated GUI functions, by using the ActiveX components layer of the document management system illustrated in Fig. 1.

It will be understood that while Fig. 1 illustrates a complete set of integration layers, it is within the scope of the invention to provide the document management system with only one interface or a subset of the interfaces shown, all dependent on the ISVs's intended implementation of the system.

Architectural model of the interfacing of the document management system and a client

This section details the architecture of the interfacing between the document management system and a client, including the applications and communications protocols that will be used. The diagram in Fig. 2 gives a graphical representation of the architecture.

The client

The user will typically have a Personal Computer (PC) connected to an Intranet. Although the system is easily modified to support the use of the Internet and Extranets with some added security, the described embodiment is Intranet based. Also, although at the moment most client devices connected to Intranets are PC's, in the future there are likely to be other "Intranet ready" devices (such as a Network Computer) connected to the Intranet.

Each Intranet client has a web browser installed. Any Java enabled browser can be used to access the system which includes the two most popular browsers at the moment: Netscape Navigator and Microsoft Internet Explorer.

The user will typically point their browser to the location of the system home page. Alternatively, the users browser could be set up to automatically point to the homepage upon start-up. The browser will then render the initial HTML page for the system that will have the initial client running within it. The user can then log in and start using the system.

As the client Java applet requires other JavaBean components it will download and execute them accordingly, ensuring that no client code need be installed on the client.

The server

The server is a separate server process running on the web server to ensure it is not integrated into any particular web server. The client communicates with the server via CORBA/IIOP.

The database server

The server application communicates with the database via JDBC. This ensures that it is compatible with a wide range of relational databases. The database may reside on the web server or elsewhere.

Component interaction

The component architecture is JavaBeans. Where it is not possible to use JavaBean components, "bean wrappers" is used to ensure that other components types (ActiveX or COM) will be compatible with the overall architecture. It is for some components necessary to use JavaScript to facilitate communication between ActiveX viewing components (i.e. components that run with the browser) and the system.

A bean is a component that has a degree of autonomy. It can be seen as a separate entity which "lives" in the system and responds to events which are sent to it and fires events to other beans as necessary. Beans are typically instantiated and "wired up" at system start-up. Most beans have Graphical User Interfaces (GUIs), though they don't have to, that are typically started up as "hidden" when the system initialises. When they are required to perform a function a message may be sent to them at which point they "show" themselves and perform the required function. This may require the bean to send an event back to the requesting bean telling it that it has finished performing its function or it may be necessary to send an event to another bean.

The formal definition of a bean is a "class which has clearly defined methods, properties and events and can also be used within a development environment" such as VisualAge for Java and JBuilder, etc. Distributed object communication is CORBA/IIOP, that is the "link" between the client and server.

Software Architecture

The diagram in Fig. 3 shows the vertical architecture of the system. It shows the overall packages for the system with their dependencies.

Packages: View and Control

The view package contains all the GUIs of the components and the control package contains each of the GUIs controllers. This design pattern is commonly known as the Model-View-Controller design pattern and is used by the new event model in Java. A good description of this pattern can be found in September '97s edition of the Java report though it is well documented in most design books including Gamma et al. This pattern also fits in well with the JavaBeans architecture that is used in the design of the system. Only the client side of the system uses these two packages.

Packages: Model: Interface, Skeleton and Implementation

The model package contains three sub-packages that contain the structure of the business objects of the system. This design is used so that the business objects can be distributed.

The interface package is the glue that holds the distributed part of the business objects and the client side of the business objects. This is the only package that exists on both the client side of the system and the server side.

The CORBA IDL compiler generates the classes in the interface and skeleton packages. The skeleton classes implement all the CORBA distributed hooks necessary to distribute the objects. The skeleton classes are super-classes of the distributed objects.

The implementation package contains the classes that implement all the IDL compiler generated interfaces in the interface package.

The broker package

The broker package is responsible for storing business objects in the database. Each business object has its own broker object that it uses to make itself persistent and "inflate" itself as necessary. The broker class contains the necessary JDBC calls to the database for the given business object.

Brief component descriptions

Archive component

The archive component allows the user to create and store a new document. A document comprises of one or more data files, one or more index cards, security and a universal identifier. It is the responsibility of the archive component to create this data using other components as required.

The first step in creating a new document is creating the document object that will own all the constituent parts that make up a document. The various parts of the

document can now be obtained by delegating responsibility to other components within the system.

The data files for the document are acquired from a data source. A data source listens for messages requesting data files. It then processes these requests, obtaining some data files and adding them to the documents data container. Different types of data sources will exist to get files from different external sources, e.g. a scan manager acquires files by scanning paper documents, a file system data source allows a user to select files from some available file system and an in-tray allows some external system to provide files which it would then add to the document. A more detailed description of these data source components is provided below. Data sources that acquire files by any other means can be added to the system as required.

The index card for the document is acquired from the index card manager component. The index card manager component responds to the request from the archive component and provides an index card for the document.

The same process is repeated to acquire document security from the security manager.

The completed document can now be committed to the system. When this has been completed successfully the archive manager sends an event to the free text manager informing it about the existence of the new document that so it can schedule it to be free text stored.

Retrieve component

The retrieve component is responsible for retrieving documents stored within the system. There are two aspects to this, searching for the documents and displaying the results of the search.

There are three different ways for a user to find a document within the system. Control of the search process is delegated to the appropriate search manager depending on the search method chosen.

Free text searching is delegated to the free text search manager. The free text search manager creates the free text logic. The GUI used to create this logic is based upon the yahoo interface as this is easy to use and familiar to most users because yahoo presently is one of the most popular search engines in use on the Internet.

The free text logic allows the user to input various words and search constraints in a very simple way, e.g. entering " + invoice + scanner -fujitsu" will find all documents that include the words invoice and scanner but do not include the word fujitsu.

The free text logic is passed to the free text manager. The free text manager returns a list of document references that match the free text logic. The list of document references is then passed to a results view as described below.

Index Card Searching is delegated to the index card search manager. This search allows the user to find documents based upon the structured index card data. The construction of the search data is performed by the index card manager component. The search is based upon any information stored within an index card, e.g. a search could be based upon an index card template or an index card search template.

The search data is passed to the index card search engine, which returns a list of documents that match the search criteria. These results are passed to the results view.

Browse searching allows the user to examine all the documents stored within the system. The documents are examined as described in the results view below.

Results view

There are two ways in which results of a search may be viewed, a summary view and a tree view. These two views will be interchangeable as required by the user.

Summary view: The results are displayed as a list of short test summaries based on information in the documents index card. This is very similar to the way yahoo displays its results.

Tree view: The results are displayed in a tree structure based upon the category fields within an index card. Categories are fields of a specific type. For example, an Invoice card may have as the first three categories on the index card: Company, Product and Payment Date. A tree hierarchy of categories will then be generated as the user stores documents. An example of what this hierarchy might look like is shown in Fig. 4.

By selecting a "tree view" the user will be presented with a collapsed tree from which she/he can expand branches which are relevant to them.

Document view

Using either form of the results views the user can select the document they wish to see. A message containing the documents reference is then broadcast so that a suitable document viewer listening for the view request can display the document.

Index Card Manager Component

This component is responsible for the creation and management of index cards. Index cards consist of a series of fields. Fields can be a mix of the following:

(a) New or Predefined
(b) Mandatory or optional
(c) List or Text

(a) New fields are fields that the user creates from scratch. Predefined fields are fields that have already been created and are in use in other index cards. An example of this may be a "Company" field that appears in the Invoice index card and Sales index card.

(b) Mandatory fields are fields that the user must enter when storing a document and filling in the index card. An error message will be displayed to the user if she/he fails to enter a value into the field. Optional fields may be left blank by the user.

(c) Text fields are fields in which the user is able to enter free text. List fields are fields that consist of a List of choices. For example, the "Company" field could be a List Field. This means that when the Company field appears on an Index Card, the user will be restricted to selecting from a list of predefined companies. The advantage of this is apparent when searching for a given company. By searching on the company name "Mitsubishi" the user can be confident that she/he will receive a list of all documents that have the field value Mitsubishi and will not miss documents in which the company name Mitsubishi has been miss-spelt or spelt differently.

Placing List Fields on an index card in a defined sequence means that stored documents can be viewed in a hierarchical manner (See Tree View Search). This makes the job of the Index Card Manager more of a system configuration role.

The index card manager also has the ability to modify index cards. Fields can be added to index cards or disabled in an index card. The Index Card Manager can also disable and enable index cards.

Directory Component

The directory component is responsible for providing data files that it obtains from a file system. A request will come from the archive component for data files to be added to a data container. The directory component will then allow the user to interact with the file system and select the files to be added to the data container.

Scan Component

The scanning component is responsible for providing data files that it obtains by scanning paper documents. A request will come from the archive component for the data files to be added to some data container. The Scan Component will then be responsible for driving the scanner to obtain these files.

The scanning interface uses the existing industry standards of TWAIN and ISIS in communicating with the scanners in order that the widest range of scanners can be supported.

In-Tray Component

The in-tray component is responsible for providing data files that it obtains from an external source. A request will come from the archive component for data files to be added to a data container. In the case of the in-tray component these files will already have been provided by some external system.

The in-tray will provide an interface to allow external systems to add files directly to it. The files stored within the in-tray may be scheduled or unscheduled, i.e. if the external process has informed the scheduler that there are scheduled files waiting to be stored or if the in-tray checker needs to poll the in-tray to see if there are any unscheduled files available to be stored.

This component is designed to allow seamless integration with external systems so that documents can be created automatically with little or no user interaction.

Free Text Manager Component

The free text manager component is responsible for ensuring that documents are free text stored and retrieved. A core part of the system functionality is the ability to search for documents based on their content. The content of a document will vary according to the type of file so the system will have to have the ability to read all common file types and be expandable to cope with new file types. These file types can be stored in very different file formats such as ASCII text in word processor documents, image data in scanned documents and sound in audio files. The technologies for reading this data can therefore be very varied. The system provides support for searching the content of most common PC office application files and TIFF Group 4 image files.

When a document is registered with free text manager component the free text manager schedules it to be free text stored. When the document is ready to be free text stored it is passed to the free text archiver to be processed.

The free text archiver navigates the document to obtain its data files. These files are then presented to the free text engine to be stored. However, in some cases the file format may be unknown to the free text engine and so these files need to be processed by the text extractor to obtain a text file, which can then be presented to the free text engine. The free text engine selected should support most file formats but the text extractor will allow the system to be extended to support other file formats such as image files where Optical Character Recognition (OCR) will have to be used to extract the text.

There are various free text engines on the market for indexing files by content. These have become more common as a result of people wanting to search Web Sites on the Internet by content. Anyone from a number of these products may be selected as the base for the free text engine for the system. Some of the factors that influence this decision are cost, supported platforms, supported file formats and ease of extending the basic functionality in order to integrate it with the system.

The free text component is based on a third party free text engine with further integration to allow seamless integration with the rest of the system. Some possibilities for the free text engine are:

- Microsoft Index Server - Free with Microsoft web server, which will be the initial supported platform for the system. Will read any Microsoft file formats and is extendable to read any other format. Will only work on Windows 32 platform.

- Alta Vista - Supports multiple file formats and has a Software Developers Toolkit (C++). Will run on Windows 32 and Unix.

- Muscat - Supports multiple file formats. It is available in Java and is therefore platform independent.

Free text engines do not currently support the ability to extract text content from an image for which reason an existing OCR is integrated with the free text engine to enable this functionality. There are various off-the-shelf OCR products available so one from a number of these may be selected.

The basic search facilities for searching for a document by content include key word logic and wild card searching, e.g. search for documents containing John and Dav*. Depending on the choice of free text engine other search capabilities may include fuzzy logic and synonym searching. Fuzzy logic searching is the ability to find words similar in appearance to the search criteria, this could be important in search for OCR'ed documents as mistakes in the recognition may cause the odd letter to be incorrect. Synonym searching is the ability to find words similar in meaning.

Annotation Component

This component allows the users to add annotations to either a data container or a data file. In the present embodiment of the system, the annotations are textual notes that are viewed separately from the document. The annotation component is responsible for the editing and viewing of the annotations.

When a page/file from a document is viewed a message will be sent to an annotation viewer requesting that the annotation is displayed. The annotation viewer will then display the annotation.

Audit Component

The audit component is responsible for the recording of any selected audible actions that occur within the system. When documents are stored by the system it is important to track their progression to enable management and auditing of the system. The management of the system may involve tracking when and how documents are stored, monitoring who is viewing the documents etc. The auditing of a document is important to validate the authenticity of the document so if required the history of the document can be shown. The scope for auditing also extends to many other aspects of the system such as monitoring user and security management,

examining throughput etc. Therefore, all classes with the system that perform actions that might need to be audited are required to send an audit event message to the audit component.

Within the audit component the audit event message is relayed to any registered audit event listeners that will then store the event in some persistent archive. Initially, there are two types of audit event listeners. The first is responsible for system management auditing, i.e. recording actions such as document creation, user management, index card management etc. The second audit event listener is responsible for more specific examination of the behaviour of components in the system. It will log behaviour at a more detailed level that can be used for diagnostics or debugging. Both audit event listeners are configurable so that they can be set up to record the occurrence of whatever actions are required.

Security Component

The security component is responsible for the management of users and document security.

A fundamental part of a document management system is the ability to restrict access to areas of the system to certain groups, e.g. account documents might only be available to people working in the accounts department and managers. The restriction of document areas becomes even more important if the document management system is going to open up documents to the Internet.

The chosen security model is simple to use and yet powerful and flexible enough to meet with most customers requirements. The security component for the system is designed to achieve this requirement but also with the added flexibility that it can be replaced with another security component so that a different security model could be introduced with little impact to the rest of the system. This may be done if either the chosen model was considered not to be adequate or a customer had a specific security requirement not met by the chosen model.

The security model is based around users and user groups. The users are organized into logical groups, e.g. departments, security levels, etc. All documents stored in the system will be considered unsecured until restrictions are applied. A restricted document will be assigned users and/or user groups that can access the document. A user will then only have access to the document if she/he is an assigned user or in an assigned user group for that document. The decision as to whether a user can see a document is made by the security component.

Index cards can have security settings associated with them. When a document is stored and first assigned an index card the associated security will become the default security for that document, e.g. an invoice index card may be assigned to a user group named ACCOUNTS so when a document is stored with the invoice index card it will by default be assigned to the user group ACCOUNTS. If any further index cards are added to the document they will not affect the security of the document.

System administration functions such as user management, index card creation, index data modification, and storage can be assigned to users in order to administer the system

Document View & Print Component

Once the stored document is retrieved the user may select files within the document to view. Depending on the file type (i.e. TIF, audio, video etc) the browser will render the file using a JavaBean component, plug-in component or ActiveX component.

Wherever possible the view components are obtained from third parties as the scope of required view components is unlimited, e.g. a Microsoft Word viewer for Word documents. The component architecture of choice is JavaBeans. This means that view components will be JavaBeans unless a component of this type is not available. If there is no JavaBean view component available for a given file type then an ActiveX or Plug-in component may be used.

The component used to view the document will also be used to print paper copies of the document as it is the only part of the system that will know how to render the documents data.

JavaScript is used to seamlessly integrate viewing components into the system.

The Log-In Component

The initial graphical user interface of the system is configurable so that a particular user will have access to areas of the system she/he is likely to use, e.g. the system could be configured to start with any archive interface if the users main job is inputting files, it could automatically perform some search and display the result set such as the previous days mail or it could simply have a control pad so the user can select what they want to do. The scope for what could be possible is almost unlimited as the whole design of the system is based around a component architecture were the functionality of the system is captured behind a component interface and the front end display can be designed on top of this.

The system provides the choice of either displaying the home page for the system or provide a control pad for accesses the functionality of the system that can be embedded in the customers own HTML pages.

Design Rationale

This section describes the reasons behind the design decisions made focusing on why the various tools languages and protocols have been chosen.

Java

Java's explosive growth since it's official release just over 18 months ago has created a great deal of interest in the computing industry. So far it has lived up to expectations of becoming the programming language of choice for Internet and Intranet based development and is endorsed by all industry leaders. It has established

itself as a one of the main programming languages in use and is predicted to be the most popular development language within two years.

The two main reasons for using Java as the main programming language in the system are detailed below:

(a) Java is platform neutral.

Java programs should be, and are to a large extent, able to run on any platform without any recompilation required. Forester Research (see www.forester.com) predicts that by the year 2000, 10% of desktop devices will be Network Computers. Also, other "Internet ready" devices are currently being developed such as PDAs (Personal Digital Assistants). By writing the system in Java it is positioned in such a way as to take advantage of these new hardware platforms.

(b) Productivity

Java, in comparison to C++ (the only other real alternative), is easier to learn. Java is also a much more readable language so it will ensure the code is much easier to maintain.

Sun Microsystems and many other companies provide a rich set of APIs to use when developing systems using Java to speed up application development.

Components

Components enable Software Engineers to create systems that comprise of individual autonomous parts. The advantage to this is that these "parts" or components can be plugged in and unplugged without too much effort. This means that components can be redesigned and rewritten easily as the system evolves, it also opens up the possibility of buying components "off the shelf" rather than "rolling your own".

There are two main component architectures to choose from: JavaBeans and ActiveX. ActiveX is Microsoft's platform specific component solution. Its complex nature and

large download times have meant that mainly sites that have a large Microsoft legacy base have adopted it.

The component architecture for the system is JavaBeans. This architecture is platform neutral as it is written completely in Java and is a natural extension of the Java language.

Distributed Components

Distributed Components are components that run on a different computer or on the same computer but execute in a different process. Because the system have components on both the client (i.e. the browser) and the server (the web server) it is necessary for these components to communicate over the Intranet/Internet.

Again, there are two main distributed component architectures to choose from: CORBA and DCOM. DCOM is Microsoft's distributed component architecture. CORBA is the Object Management Groups distributed computing architecture. The OMG comprises of the main computing companies excluding Microsoft. Java integrates with CORBA well and is supported on all major hardware platforms so the Distributed Components choice for the system is CORBA.

Database Components

There are two main options as to the type of 3rd party database that could be used as the core part of the database component, relational databases and object databases.

The technology and products available for relational databases are very well established and accepted as a market standard. Just about any company will have an RDBMS and will be entirely happy with the technology.

The type of data the database will be expected to store does not readily map onto a relational database therefore additional development will be required to convert this data to the required format. However, there is an increasing number of products on the market to aid this process.

The JDBC standard exists as a method of interfacing to different databases with the same application code e.g. Microsoft, Oracle, Sybase and Informix.

Although object databases are now a fairly mature technology the database products have still not been widely accepted and generally have a bad reputation as regards reliability and speed (which may or may not be the case!). The customer base for object databases is very limited. The storage of application data would however be straightforward and potentially could have large savings in development time. Although there is a market standard for interfacing an object database (ODBMG) it has only recently been accepted. This means that the choice of database would be limited.

Although there are some major advantages for using an object database its lack of marketing maturity is too important to overlook, and therefore the present embodiment of the system is based on a relational database. However, the system may easily be modified to comply with the object database standards and the maturity of the object database market is likely to increase considerably over the next year or two.

Detailed description of components

A number of the components of the system are in this section described in details.

Archive component

The archive component is concerned with the creation and storage of new documents. A document comprises of one or more data files, one or more index cards, security details and a universal identifier (OID). The archive component will request this data from other components whereafter it commits the completed document to the system.

The archive component (Archive Manager) will listen for events, which correspond with creating new documents, populating those documents and storing the documents. These events are generated by user interaction and the process is explained in more detail below.

The system executes in a browser frame window and the archive process is activated in response to a click on a GUI component such as a menu item (i.e. the standard File|New menu command). The archive component will then generate events for other system components to obtain the various document parts. Requests will be made to obtain index card(s), universal identifier, files and security details for the document. The appropriate system components will supply this data by either returning it directly or loading their own GUI and prompting the user for input.

The archive component manages the interaction between the other components and the user. When a new document is created it is mandatory for it to have at least one index card and data file, as well as a universal identifier and security details. Therefore, a user creating a new document should be prompted through the various component interfaces to input the necessary data. The archive component generates events to control the interaction and provide a simple GUI component (such as a dialogue box) with Ok and Cancel buttons at the end of the archive process to either commit the completed document to the system or abort the process without storing. If the document is to be committed the archive component sends an event to the Free Text Manager for the document to be free-text stored. There may also be the need for the archive component to provide feedback for the user during the process.

The archive component acts to co-ordinate the creation and storage of a new document. The component is a single bean that listens for user-generated events and responds by firing events to other system components, which populate the document. The user-generated events are instances of clicks on menu items or buttons. The Archive Manager detects these events as it is wired to the window frame in which the system is executing. Fig. 5 represents the interaction between the Archive Manager and other system components. User events are generated by interaction with the window frame and as mentioned previously, the Archive Manager is wired to respond to those events. The Archive Manager fires events requesting index card and security information and listens for events in response to those requests. It also requests data files for a document, initiating the documents data container. The archive process is complete when a user event signals the Archive manager to commit the document to

the system and generate an event for the Free-Text Manager to free-text archive the document.

The class diagram shown in Fig. 6 models the relationships between the Archive Manager and the other components and classes concerned with the storage process. The Archive Manager is wired to specific events generated by the window frame, the frame is created by the users web browser when the system is executed. Archive Manager is therefore able to respond to system events such as mouse clicks on menu-items or other GUI components owned by the window frame.

The Archive Manager can respond to system events to initiate the creation of a new document and to confirm if a newly created document is to be stored or discarded. The sequence diagram in Fig. 7 shows the messages and events that occur when a user chooses to create a new document and commit it to the system. The NescGUI class creates the frame run by the system. This class is responsible for creating the system components and adding listeners between those components. Archive Manager responds to a user event to create a new document (1) by calling the system component constructors and wiring the components through their listeners (2-17). The archive manager constructs the business objects such as the document and its data container and makes requests to other components to complete the document structure. The system components communicate through adapter classes that act as a bridge between the components. A component will fire an event and the adapter class will respond to that event by invoking an appropriate method in another component. This removes all dependencies between the system components.

The Archive Manager fires an event for the DataSourceFactory class requesting data items for the documents data container (18). This is dealt with by the ArchDataAdapter class, which invokes the populateDataContainer method in the DataSourceFactory (20). The DataSourceFactory will present a user interface, which will depend on the data source type and allow files to be added to the data container. The DataSourceFactory will fire an event to the ArchDataAdapter when the data container is complete (21) which will call a method in the Archive Manager (22) to continue the storage process.

When the data container has been filled, the archive manager requests an index card and security for the document by firing events to adapters for the index card manager and user manager and responding to events fired from those components, via the adapters (23-32). At this point the document structure is now complete and the Archive Manager displays a GUI component to prompt the user to commit the document to the system or abort the process (33). In this case the user chooses to commit the document (34) and calls on the document to invoke its commitDocument method.

The Archive Manager completes the archive process by firing an event to the Free Text Manager component registering the document for free-text storage (37-38).

Retrieve component

The retrieve components function is to find and display documents stored by the system. The component consists primarily of two beans, a retrieve manager and a search manager. The search manager class is abstract with the implementation of the class dependant on the type of search carried out. When a search has been carried out, the search manager will have produced a result set which contains information from documents that match the search criteria. The user can then choose to have the result set displayed in either a summary view or tree view format, then select to view a documents content with an appropriate viewer.
The components in the retrieve process communicate via events so there is no dependencies between the classes.

The class diagram shown in Fig. 8 models the relationships between the Retrieve Manager and the other components and classes concerned with the retrieve process. The Retrieve Manager reacts to user events generated by the main frame GUI.

The retrieve manager fires events to allow the searching and retrieval of documents and co-ordinates the display of documents chosen by the user for viewing. The sequence diagram shown in Fig. 9 outlines the messages and events generated for each type of search that can be carried out during the retrieval process:

1. User generated retrieve event (e.g. mouse click on menu item)
2. Main frame GUI responds to 1 by requesting GUI component with search options
3. User chooses index card search
4, 5. Event fired to initiate search
6. GUI prompting user to enter search data
7. Search engine passed user search data
8. Search engine compares user search data with index card data to return references to matched documents
9. Search manager receives list of matching documents (DocumentRef contains the OID, Title and summary information)
10, 11. The document data is passed to the retrieve manager
12, 13. Retrieve manager passes ResultSet and GUI component to the result manager
14. Result manager formats ResultSet data for display in panel
15, 16, 17. Retrieve manager passes results panel for display of document summary view to user
18, 19, 20, 21. User requests to change current view, as current view is summary, display changed to tree view
22, 23, 24. User selects document from results, documents OID is passed to users session object on server to retrieve document contents
25, 26. Retrieve manager requests view component to display the document using an appropriate viewer.

The sequence diagram shown in Fig. 10 represents a free text search. The implementation of the search manager and search engine classes is specific to free-text searching and the retrieve process is exactly the same after the search engine has returned the ResultSet. This situation also applies in the sequence diagram shown in Fig. 11 for browse searching. Browse searching allows the user to view any document in the system through a tree structure. The first two search types will return a set of matching documents which will not change until another search is carried out, however browse searching will present the user with a top level tree view of documents in the system based on their index card data. Each time a user expands or closes part of the tree the result set will change. Therefore, in this case the search data will be continually updated as the user navigates through the document tree structure.

The sequence diagram shown in Fig. 12 represents the initial stages of a browse search. However, as previously mentioned, the search engine will update the information as required to synchronise the result set with what is displayed on the user interface.

Index Card Component

When a document is to be stored, it must be stored with information that can be used to retrieve the document. This information is entered into an Index Card. An Index Card is a series of predefined entry fields, which indicates to the user what information is required to archive the document.

The Index Card component has three main functional areas. The first is in the creation of the Index Card Template - creating the actual Index Card itself. The second is indexing a document which involves entering information into the index card. The third functional area involves viewing the filled in index card when retrieving a document from the archive.

The Index Card Manager is the "gatekeeper" of the index card component. It is a JavaBean that listens for Index Card events. Its purpose is to field all events for the Index Card Component and pass the events on to the appropriate bean within the component. This hides the internal interface of the component to the "outside world" so it can be modified and reused without effecting any other components in the system.

The bulk of the functionality of the Create Index Card Templatecomponent is taken up in the creation of Index Card Templates. The administrator of the system will create Index Card Templates. This will ensure that there is not a confusing array of Index Card Templates to choose from when indexing a document.

After naming the Index Card Template, the administrator will then add Index Card Field Templates to it. Index Card Field Templates (ICFT) will initially be of two types: List Fields and Text Fields.

The administrator has two choices when adding ICFT's, she/he can either add an existing ICFT to the template or create a new ICFT and then add it to the template. If the administrator chooses to add a new ICFT to the template then that ICFT will be available for inclusion in other templates.

Creating Text Field Templates is very straightforward. The administrator simply decides on the name of the label and width of the text field and then indicates via a radio button if the field is optional or mandatory.

Creating List Field Templates requires a further step. The administrator must create the List Items or choose existing List Items to add to the List Field. If the administrator creates the List Items they will be available to add to other List Fields in the future.

After completing the creation of the Index Card Template, it will be stored in the database and retrieved when indexing documents or modification of the Index Card Template.

When the user wishes to index a document she/he will be presented with a list of Index Card Templates to choose from. The user selects a given template and then proceeds to enter information into its fields.

The information entered will be stored with the document for retrieval and possible modification at a later date.

When a document is retrieved from archive the user has the option of viewing the index information of the document. If the user chooses to view the document an event will be sent to the Index Card Manager to pass on to the appropriate Index Card View bean.

The Index Card Component is a composite bean consisting of nine beans. The flow of control within the beans is shown in Fig. 13. The beans communicate with each other via events so there is no direct dependency between their classes.

ICTemplateCreateGui is the main bean for creation of Index Card Templates. The beans which are used to create templates are: ICTextFieldTemplateCreateGui which allows the user to create Text Field Templates to add to the index card; ICListFieldTemplateCreateGui which allows the user to add List Field Templates; ICListItemsCreateGui and ICListItemsListGui is used by ICListFieldTemplateCreateGui to create List Items and select them respectively; ICFieldTemplateListGui is used to select existing fields to add to the Index Card Template.

ICViewGui is responsible for displaying the index card. ICIndexDocumentGui allows the user to fill in index card information so that documents can be stored.

The sequence diagrams shown in Figs. 14-16 explore different aspects of creating an Index Card Template. Also, the sequence diagrams in Figs. 18-21 show how the beans and GUI controllers are wired up so that they can send messages to each other. The class diagrams generated from the sequence diagrams are also presented.

The sequence diagram shown in Fig. 14 models "Creating an Index Card Template and adding a new List Field" (CreateICTNewListFieldI). The ICTester class (1-3) is used in the sequence diagram for testing purposes, and the ICManager functionality is activated by bean events when the system is installed.

The ICManager is, as mentioned earlier, the gatekeeper of the components and all event requests come through it. ICManager calls activate() on ICTemplateCreateGui (4) which has already been instantiated when the beans were wired up. activate() simply displays the ICTemplateCreateGui and passes control to the user to choose a field template type to add the Index Card Template.

In the shown example the user decides to add a List Field Template so ICTemplateCreateGui creates the required event to fire onto ICListFieldTemplateCreateGui (5-7).

ICListFieldTemplateCreateGui creates a new ICListField object and allows the user to enter its label (8, 9). The user then requests to create and ICListItems object to add,

ICListField TemplateCreateGui creates the necessary event and fires it off to be caught by ICListItemsCreateGui (10, 11).

ICListItemsCreateGui then allows the user to enter list items whereafter it fires an event object signifying that the ICListItems object has been created. This event is caught by ICListFieldTemplateCreateGui as it has been wired to listen to these events. (12-17).

ICListFieldTemplateCreatGui then generates and event signifying that the creation of the ICListField object has been completed. This event is caught by the ICTemplateCreateGui that displays the ICListField using ICFieldTemplateDisplayGui (17-21).

Finally, ICTemplateCreateGui calls commit() on the ICTemplate to make it persistent.

The sequence diagram shown in Fig. 15 models "Creating an Index Card Template, adding a new List Field and selecting a predefined List Items" (ICTListFieldII). This sequence diagram is almost identical to the sequence diagram shown in Fig. 14 except instead of creating a new ICListItems to attach to the ICListField an ICListItems is selected from a list (11-14).

The sequence diagram shown in Fig. 16 models "Creating an Index Card Template and adding a new Text Field" (CreateICTNewTextField). Again, this sequence diagram is similar to the sequence diagram shown in Fig. 14. In this scenario the user chooses to add an ICTextField to the ICTemplate. (8-12)

The class diagram shown in Fig. 17 models "Creating an Index Card Template and adding a new List Field" (classdiagrams.createICTNewListField). This is the class diagram for the sequence diagram shown in Fig. 14. ICTextField and its related classes have been omitted from the diagram in order to portray the design clearly.

The sequence diagram shown in Fig. 18 models "Wiring the Create Index Card JavaBeans" (createICTWireBeans). This sequence diagram shows how the beans are

wired up in the creation of the Index Card Template. The beans will be wired up upon initialization of the system so that they can then send events to each other.

The sequence diagram shown in Fig. 19 models "Wiring the Controllers to their GUIs" (createICTWireControllers). This diagram shows each of the GUIs controllers being wired up to listen to events from the GUI as defined in the Model-View-Controller design pattern.

The class diagrams shown in Figs. 20 and 21 model "Wiring the Create Index Card Template JavaBeans" (classdiagrams.createICTBeanListeners & createICTBeanEvents). These two class diagrams show the relationship of events, event sources and event listeners in the creation of an Index Card Template. These diagrams show how the event model is used in the system components. See "Mastering JavaBeans" for a detailed explanation of JavaBeans and the 1.1 event model. The diagrams show how the beans are wired together so that they can listen to appropriate events that they must field. The second class diagram concentrates on the wiring of the controllers to their GUIs. These controllers are responsible for listening to general GUI events such as requests for on-line Help.

The sequence diagrams shown in Figs. 22 and 23 model "Commit Index Card" and "Populate Index Card" (CommitIndexCardI&II & PopulateIndexCardI&II). These sequence diagrams show in more detail the broker architecture. Each persistent object in the index card implements the Committable interface. This forces the class to implement the commit() method which starts the process of committing the object to the database.

When a new persistent object is instantiated the constructor will automatically allocate the object a unique object identifier (OID). Each persistent object will have a constructor that takes as its parameter an OID. When this constructor is called it will "inflate" the object from the database using its Broker object. Each persistent object has its own broker object. The Broker object simply takes the OID in its constructor and calls the JDBC SQL required to populate the object. The business object then interrogates its broker object for the information it requires to populate itself.

The broker object is responsible for the storage and retrieval of the persistent object. Each broker class inherits from the abstract class Broker that provides database connections using the DBConnectionPool class.

DBConnectionPool pre-starts a number of connections to the database. These connections are then dispensed and released by class methods, enabling an object from any part of the system to gain a connection to the database.

The class diagram shown in Fig. 24 models "Commit and Populate Index Card" (classdiagrams.Persistence). This diagram shows the persistent classes and broker classes and their relationships with the Committable interface and Broker abstract class.

Note that there is no dependency from the broker class to its persistent class.

The sequence diagram shown in Fig. 25 models "Index the document" (IndexDocument). This sequence diagram goes through the steps of selecting and Index Card Template and allowing the user to enter data into each of the fields of the template.

The sequence diagrams shown in Figs. 26 and 27 model "View the Index Card" (ViewIndexCardI & II). These sequence diagrams go through the steps of displaying index information for a given document.

The following section describes the implementation of the Create List Items bean to demonstrate in Java code both the vertical architecture and infrastructure of the system.

The only package not represented in the Create List Items bean is the controller package. ICListItemsCreateGui does have an associated ICListItemsCreateController. This class is responsible for responding to requests for system help and error messages, etc. with internationalisation.

The purpose of the Create List Items bean is to create new ICListItems objects as shown in details in the sequence diagram in Fig. 14. The bean allows the user to enter a name for the object and then allows the user to enter items to be associated with the ICListItems object. Once the user has completed entering the entire list items she/he enters OK and the ICListItems will be stored in the database via CORBA and JDBC. At this point control will return back to the ICListFieldTemplateCreateGui object.

Before storing the ICListItems object the bean checks that the name of the list items does not already exist. If it does, a message is displayed to the user.

The class diagram shown in Fig. 28 models the "Create List Items bean". This class diagram shows the classes involved in the Create List Items bean. Each of the packages (except controller) is represented and it shows the vertical architecture of the system in action. The Server object is responsible for starting up the ICListItemsDispenser distributed object. It names the object and publishes it to the ORB so that it is ready to take method calls.

ICListItemsCreateGui is the GUI for the bean. It implements the ICLstItmsCrtRequestListener listener interface as it will be listening to ICLstItmsCrtRequestEvents from ICListFieldTemplateCreateGui.

The constructor for ICListItemsCreateGui calls instantiateICListItems() (via initialise) which binds to the ORB and locates ICListItemsDispenser. It then calls instantiateListItems() on the dispenser object to get an interface reference to an ICListItems object which it can then use to populate the object.

The two interfaces ICListItemsDispenser and ICListItems are generated by the IDL-to-Java compiler and form the "glue" between the client and server sides of the application.

The two skeleton classes _sk.ICListItemsDispenser and _sk.ICListItems are also generated by the IDL compiler and contain the CORBA plumbing interfaces.

When commit() is called on the ICListItems interface the commit() method of the implementation class is called on the server via the ORB. This then invokes the commit() method of the ICListItemsBroker object which gets a database connection from DBConnectionPool and executes the update JDBC SQL statements to commit the object to the database.

Directory Data Source Component

When a user wishes to add files to a document, she/he will be presented with a choice of Data Sources from which to retrieve files: In-Tray; Scan Manager; Application Data Source and Directory Data Source. The Directory Data Source component is responsible for retrieving files from the local drive and adding them to a Data Container.

The Directory Data Source component presents the user with a tree structure reflecting the contents of their disk drive. The user is allowed to parse the directory tree and select files to be added to the document.

The Directory Data Source Component is a composite bean consisting of four beans. The flow of control within the beans is shown on Fig. 29. The beans communicate with each other via events so there is no direct dependency between their classes.

The class diagram shown in Fig. 30 models the Directory Data Source Component. The Archive Manager, Data Source and Converter objects are not part of the DDS Component but are included in this diagram to provide context within the whole System Detail Design.

The DDSDirectoryBrowser object encapsulates the GUI that allows the users to parse the directory tree and select files which are to be added to the archive.

The DDSDirTree object displays the contents of the current drive in a tree structure. When the user selects a branch of the tree (a directory) an event is fired to all listeners.

The DDSDirLister object displays the contents of the current directory. If the user selects an object (either a file or a directory) then an event is fired to all listeners. It listens for events from itself and from DSSDirTree to indicate that the current directory has changed and it should update its view to reflect the new directory.

The DDSBasket object contains a list of handles to files which the user wishes to add to the archive. It listens for events from the DDSDirLister object to indicate that a file is to be added to its list. Furthermore, is listens for events from itself indicating that a file is to be removed from its list.

The sequence diagram in Fig. 31 shows the construction of the objects and how they are wired up to listen for events.

The sequence diagram in Fig. 32 shows the interaction between objects as the user parses the directory tree to select a file to add to the archive.

1. When the user selects a directory within DDSDirTree, DDSDirTree calls fileSelected(FileSelectedEvent) on DDSDirLister to indicate that the currently selected directory has changed.

2. DDSDirLister gets the currentDirectory from the FileSelectedEvent it received, gets the contents of that directory, sorts the files and displays them.

3. When the user double clicks on a directory within the DDSDirLister view, DDSDirLister invokes directoryClicked(DirSelectedEvent) on DDSDirTree to indicate that it needs to update its view.

4. DDSDirTree gets the current directory from the DirSelectedEvent it received. It builds a list of directories under the selected directory, updates it view and then.....

5. .....invokes fileSelected(FileSelectedEvent) on DDSDirLister to indicate that the currently selected directory has changed.

6. DDSDirLister gets the currentDirectory from the FileSelectedEvent it received, gets the contents of that directory, sorts the files and displays them.

7. When a user double clicks on a file within the DDSDirLister view, DDSDirLister invokes filesChosen(ChosenFilesEvent) on DDSBasket to indicate that the selected file needs to be stored.

8. DDSBasket gets the files which needs to be stored from the ChosenFilesEvent, updates its list of files to be stored and displays its new list.

9. When the user hits the archive button, the DDSBasket receives the ActionEvent and then.....

10. .....Invokes filesChosen(ChosenFilesEvent) on DDSDirectoryBrowser.

11. DDSDirectoryBrowser gets the handle of each files required to archive from the ChosenFilesEvent. It then converts each of these files into a stream of bytes and then.....

12. .....Invokes filesReadyForArchiving(ChosenFilesEvent) on DataSourceAdapter which.....

13. .....Then extracts the list of files (each now held as a stream of bytes) from the ChosenFilesEvent and stores them in a vector. It then.....

14. .....Passes them to the ArchiveManager with the method dataItems(Vector).

The External Associations and Data Formats component will be used by the archive component to retrieve files. Files which are to be stored will be converted and stored as streams of bytes.

Scanner Component

The scanner component is concerned with the scanning of paper documents and handing the scanned images to the data storage component. The component is able to employ multiple scanners connected to the network. The interface to the scanner (drivers) uses the TWAIN standard, thus allowing the use of off-the-shelf scanner drivers and libraries. On the user side, a third-party application is used to display a proof of the scanned image. The image data itself will be stored in TIFF format.

The component is split into three parts: the Controller, running on every computer connected to one or more scanners, the User in the users web browser, and the Server running on the server computer. The three parts communicate via proxies, implemented using CORBA.

The client-side part, the User interface, will listen for events from the Archive Component (requesting to fill a DataContainer). The scanner-side part, the Controller will listen for events from the Server, requesting the scan of a paper document. The Server serves as network-wide mediator for the communication between the clients and the scanners. All scanners register themselves on start-up with the Server, so the Users can request a list of all known scanners and select one to use. As the Controller has to be implemented in C/C++ to access the scanner, this three-tiered approach is easiest to do. Additionally, this also solves problems concerning the tunnelling of CORBA requests through firewalls, and will even allow the remote control of scanners.

Whenever the User receives a request from the Archive Component to scan a document, it opens a user interface for the user to select a scanner to use. For this, the list of available scanners is requested on every occasion from the Server. After the user has selected a scanner, a request is sent to the associated Controller. If the scanner is in use at that time, the request is denied and the user may try again later. If the scanner is available, the scanning proceeds. Some scanner drivers may insist on opening their own user interface. In this case, the User has to run on the same computer the scanner is connected to, and logically these scanners won't be available for selection on other computers.

Together with the request, the Controller also receives the DataContainer to fill in, so upon completion of the scan, a mere signal needs to be sent back to the User. The

User now displays the image for the user to check the quality of the scan. At this point it is possible to rescan single images, delete single images, add single images, or run an additional batch scan.

The central parts of the Scanner component, the User interface, the Server and the Controller, act together to co-ordinate the scanning of documents. User interface and Server are beans that listen for events and respond by firing events to other system components. The Controller is an application written in C/C++ in order to interface with the native TWAIN-conformant scanner drivers. The incoming events are both user-generated events and system events. The user-generated events are results of mouse clicks in the user interface GUI panel, which is created by the User on demand of the Archive component and wired by the latter into the users browser. System-generated events are sent by the Archive Component, requesting the scanning of a new document. Fig. 33 presents the various parts of the Scanner component and their interaction among each other and with some other system components.

The central Server acts as messenger between the User interface and the Controller application. It passes
Requests back and forth between the two, and maintains a list of all available scanners.

The User interface applet waits for events from the Archive component. Upon receipt of an event (requesting the scanning of a document) the User interface opens a GUI in the users browser, requests the current list of available scanners from the Server, and displays it in the GUI. After selection of a scanner by the user, the User interface sends a request to scan to the selected scanner. The scanner may deny the request, in which case the interface prompts the user to retry or abort the operation. If the scanning request is successful, the scanned images are stored in the DataContainer passed with the request. After completion, the images are displayed to the user (using a third-party display utility), offering to delete images, rescan them, or to accept them as they are.

The Controller application waits for scan requests from the server and processes them. The scanners are controlled using the TWAIN standard, the drivers themselves

are off-the-shelf products. Upon start-up, the Controller application registers the connected scanner(s) with the Server, so the latter always has an up-to-date list of available scanners. The information in this list includes whether the scanner driver opens its own GUI or not. In the former case, the scanner will be marked for 'local use' only; that means that the User interface has to run on the same computer the scanner is connected to.

The class diagram in Fig. 34 models in greater detail the inner structure of the Scanner component. Though the classes ArchiveScanAdapter, DataContainer and DataFile are listed in the diagram, they are not genuinely part of the Scanner component. Instead, they are provided by other components and form the interface between the Scanner and the Archive component.

ControlListener defines the interface between Controller and Server, which is implemented using an adapter. Likewise, ServerListener defines the interface between User and Server, again implemented using an adapter. Though depicted as single classes, the adapters can (and in case of the CORBA proxy implementations do) consist of several classes on their own.

The scanners are described using instances of ScannerSpec. Every Controller creates an instance for each scanner it controls, fills it in and copies it to the Server. The server extends the instances with internal data (making them instances of ScannerData) and keeps them in a list.

Instances of JobData describe every ongoing scanning request, including a reference to the DataContainer for the image scanned. The Server keeps a list of all ongoing scan jobs, holding the JobData instances until the scan job is complete and the data is requested back from the User.

Fig. 35 is a class diagram that explores the typical inner structure of the CSAdapter and its relation to the Server and Controller class of the Scanner component. It contains all the necessary classes to constitute a proxy between Controller and Server, using CORBA for communication. As the Controller side of the proxy is

implemented in C/C++, control flow (indicated by dashed arrows) can be bi-directional.

The CSAdapter actually includes two CORBA proxies, due to its bidirectional nature. The interface between Controller and Server is defined by the two interfaces ServerListener and ControlListener. These are defined in the projects IDL file, and are compiled into two sets of Java interfaces each: ServerListener and ServerListenerOperations, and ControlListener and ControlListenerOperations. The -Operations interface is implemented by the classes CSClient and CSServer, which are the actual proxy objects.
Due to typing reasons, the Server is not connected directly to CSClient, but instead over _tie_ControlListener. The CORBA-Server part for incoming calls is take over by ControlServer and ControlClient.

The initial connection is triggered by the CSServer on the Controller side. It discovers the ControlClient on Server side using the normal CORBA naming services. Once the ControlClient is found, the CSServer calls the connection method and passes its own IOR as parameter. The ControlClient creates a new CSClient, which then uses the passed IOR to connect back to the ControlServer.

Fig. 36 is a class diagram that shows the typical inner structure of the USAdapter and its relation to the Server and User class of the Scanner component. It contains all the necessary classes to constitute a proxy between Users interface and Server, using CORBA for communication. In contrast to CSAdapter, the Server takes the role of the CORBA server. Everything else is analogous to CSAdapter.

Initialization differs between CSAdapter and USAdapter. The CSAdapter honours the fact that it is going to be contacted by non-Java clients by implementing the ControlClient the usual CORBA way. It is instantiated by the server hosts global arch.Server class and does all the necessary CORBA protocol on its own. USAdapter on the other hand utilizes the Dispenser/Session mechanism: every client application connects to the dispenser running on the server computer. The dispenser provides the client with a session, which is able to instantiate the various components of the

system. The UserServer is one of these components, clients asking for it will be handed with the CORBA generated stub class _st_UserListener.

In the current implementation, the Server initializes the ORBs for both the UserServer and the ControlServer.

The structure presented in Fig. 36 is heavily based on the implementation structure used by VisiBroker (part of JBuilder). Other CORBA packages use slightly different classes and classnames, but the same basic layout.

The important data of the Scanner component is held in the classes JobData and ScannerSpec. JobData holds the data of one scan job; ScannerSpec describes one particular scanner.
The content of these two data classes is illustrated in Fig. 37.

The JobData instances contain a reference to a DataContainer which is where the scanned images are stored (in TIF format). The 'status' field informs about the job: 0 for success, -1 for a rejected job (because the scanner is busy), -2 for a job in progress, or >0 for true errors. Field 'msg' holds a more detailed explanation of the 'status' field. 'jobId' and 'scannerId' are used to identify both job and used scanner. The ScannerSpec instances are generated by the Controller, which fills in the field 'name' with the name of the scanner, 'id' for local identification, and the flag 'bLocalOnly' with the appropriate value (other capability-oriented fields may be added later). It is copied to the Server, which keeps it in a list and adds values for the other fields: 'host' and 'scanner' to identify the host the scanner is connected to, and 'serverId' with a unique numerical value.

The action sequence of importance is the scanning of a document. The sequence diagram in Fig. 38 models the sequence "Scan Document". The sequence is triggered by the Archive component in order to scan and store a document. It does this by sending the event 'scanDocument' to the User instance, passing a DataContainer to store the data in.

The first action is that User requests the current list of available scanners from the Server. This list is then displayed to the user (not shown in Fig. 38) so she/he can select a scanner. The capability information stored in the list entries can be used to filter the list, e.g. by not displaying all those scanners that need local attention. Once the user selects the scanner and issues the scan job, the User instance sends a 'doScan' event to the designated Controller. This controller is identified by the 'id' given in the ScannerSpecs requested before. The Server uses this id to route the event to the correct Controller.

The Controller can reject the job if it is busy at the moment. In that case, the whole process of selecting a scanner and issuing the request has to be repeated, or the scanning process itself can be terminated. If the Controller accepts the job, the User just gets a confirmation that the scan is under progress. The Server will create and keep a JobData instance to track the progress of the scan. The ScanController will scan the documents and fill in the DataContainer with the TIFF compressed images. As soon as it is done, the Server is informed. Meanwhile, the User polls the Server about the current status of the scan. If desired, the User can retrieve the currently latest image while the scan is in progress to give a feedback to the user. When the scan is complete, the User can retrieve the complete JobData instance from the Server.

The User then enters the 'Image QA' phase, where the user can review the scanned images, delete single images from the DataContainer, add or replace single images, or start another batch scan process to add more images. Deleting images is implemented as one DataContainer method call, and starting another batch scan process follows the same sequence as the initial scan. Adding or rescanning a single image follows the simple chain sequence shown in Fig. 39.

In contrast to the asynchronous batch scan process, scanning a single image is a synchronous call. The Controller is passed a DataFile reference to modify, and the call returns immediately.

The Scanner Component manages the processes of scanning a document using a scanner. The component will receive events from the Archive component requesting

services, and send events to an Archive sub-component to store document data. While processing a scan request, the component will interact with the user by means of a GUI set up in the users browser window.

The design of the Scanner Component is marked by the need to interface native scanner drivers with C/C++ API with Java components. The use of a three-tiered structure solves this problem and also reduces security-problems due to firewalls intercepting CORBA messages.

Audit Component

When documents are stored in the system it is important to track their progression, thus enabling management and auditing of the system.

The main function of the Audit component is to listen for Audit events, which are generated by various Audit event sources. These Audit events are then stored in the Audit database.

The Audit Event Manager is the main JavaBean of the Audit component. The purpose of this bean will be to listen for Audit events generated by a number of sources. The bean will then archive the details of a generated event in the Audit database. The Audit component is a single bean that responds to events which are fired at it from other beans within the system.

The class diagram in Fig. 40 shows the types of objects present in the Audit bean and the relationships between them.

The sequence diagram in Fig. 41 explores the steps that are taken when an AuditEvent is generated and fired at an AuditEventListener.

Security Management Component

The system needs some form of security management in order to control access to documents once they are stored. The main purpose of Security Management is to

allocate security to a document. This involves creating a list of users and user groups who have access to the specific document.

When security is added to a document, the user is presented with a list of users and user groups, they then select the users/groups that are allowed access to the document. If the user is a system administrator they will have extra options which will include adding/removing/modifying users and groups.

When users archive a document they will have the option of modifying the security associated with it, if the user chooses not to add security the document will be given a default security setting, based on an Index Card. After the document has been stored users will not be able to modify the security settings, however administrators of the system will have that option.

The diagram in Fig. 42 shows the basic architecture of the component. The class Security Manager acts as a gatekeeper to other objects on the sever, it is responsible for creating security, groups and users, as well as adding security to a document.

The class diagram in Fig. 43 shows the main types of objects present in the Security Management Component and the relationships between them.

The sequence diagram in Fig. 44 explores the events that occur when security is added to a document.

1. createDocSecurity() is called on the User Manager from the ArchUserSecurityAdapter. It passes the document that the security is to be added to as its argument.
2. A new SecuritySelecterGUI is created.
3. The GUI asks for a user list from the User Manager.
4. The User Manager gets the user list from the Security Manager which is based on the server.
5, 6. The same process is then carried out in order for the SecuritySelecterGUI to obtain a list of groups.
7. SecuritySelecterGUI calls setSecurity on the User Manager.

8. The User Manager calls setSecurity on the Security Manager, it passes across the lists of users and groups which are to be contained within the Security class.
9. Security Manager creates a new Security class.
10. The Security is set on the document.

The sequence diagram in Fig. 45 explores the events that occur when the security is modified on a document.

1. User Manager gets a reference to the security object within a document. (passing the document as it's argument)
2. The request from the User Manager is handled by the Security Manager, which calls getSecurity() on Document.
3. User Manager creates a new SecuritySelecterGUI.
4. The SecuritySelecterGUI requests a user list from the User Manager, the User Manager obtains this list from the documents security class.
5. The SecuritySelecterGUI requests a group list.
6. The User Manager set the security setting on the SecuritySelecter GUI.
7. The SecuritySelecterGUI calls setSecurity on the User Manager, it passes across the list of users and groups that have been selected in the SecuritySelecterGUI.
8. A call is made on the Security Manager to set the security, passing the list of groups and users.
9. The Security Manager creates a new security class.
10. The Security Manager sets the security on the document, passing the newly created security class as its arguments.

The sequence diagram in Fig. 46 explores the events that occur when a new user is created on the system.

1. User Manager creates a new UserGUI.
2. The UserGUI calls setUser() on User Manager, passing user information that was obtained from the GUI.
3. User Manager calls createUser() on the server side Security Manager.
4. The security Manager creates a new user.
5. The Users commit method is called.

The sequence diagram in Fig. 47 explores the events that occur when the details of a user are modified.

1. User Manager creates a new UserGUI.
2. UserGUI requests a list of all current users.
3. User Manager requests the list from the Security Manager.
4. The users details are set in the UserGUI.
5. Once a user has been modified within UserGUI setUser is called on the User Manager.
6. The setUser call is then made on the Security Manager.
7. Security Manager makes the setUser call on the user object that has been modified, changing the fields within it.

The sequence diagram in Fig. 48 explores the events that occur when a new group is added to the system.

1. User Manager creates a new GroupGUI.
2. GroupGUI request a list of all users of the system.
3. User Manager gets the list of users form the Security Manager
4. The group details are set in GroupGUI.
5. GroupGUI calls setGroup on User Manager passing the list of users as its argument.
6. User Manager then calls createGroup on the Security Manager.
7. Security Manager creates a new group object on the server.
8. Group's commit method is called.

More user levels may be added to the security component. In the shown embodiment, the component deals with two user levels, user and administrator, however levels such as observer (read documents only) or archiver (write only) may be implemented if needed, depending on the working environment of the system.

Internationalisation / Resource Management

When displaying strings within the GUI of the system it is important to take full advantage of Java's internationalisation features. Taking this approach will avoid multiple versions of the software and enable other languages to be easily added to the systems library.

The idea behind the present resource management is that a client requests a string from the server, the string that is requested depends on the location of the client machine (or the desired language of the user). This is achieved using a server based Resource Manager which is called from the client.

The Resource Manager Server is an object which resides on the server, it deals with requests from Local Resource Managers. The Resource Manager Server is responsible for the retrieval of strings using an object of type ResourceBundle (java.util).

The Local Resource Manager resides on the client, it determines the language that is to be used and requests a string from a Resource Manager Server.

The Resource Management Component consists of two main objects which communicate via CORBA. The diagram in Fig. 49 shows the basic architecture of the Resource Management component.

The class diagram in Fig. 50 shows the types of objects present in Resource Management and the relationships between them. This diagram shows as example classes for: English (US), English (UK), and French. More could be added as needed.

The sequence diagram in Fig. 51 explores the steps that are taken when a GUI requests a string from a Local Resource Manager.

1. A GUI calls getString on the Local Resource Manager passing the key of the string that is required.

2. The Local Resource Manager calls getResource on the Resource Manager Server (via CORBA). It passes language, country (which Local Resource Manager determines) and a key, which is passed to it from the GUI that requested the string.

3. Resource Manager Server calls getBundle, a static method that takes the path of the resource bundles and a locale and returns the appropriate resource bundle, e.g. a call such as this:

getBundle("COM.mdisystems.arch.NescResources", new Locale(en,UK));

Would return the resource bundle type

NescResources_en_UK.

4. Finally, a call is made to the appropriate resource bundle, which returns the string that corresponds to the key it is passed.

The Local Resource Manager may be modified so as to allow caching of frequently used strings on the local machine, thus speeding up the system.

CLAIMS

1. A software application system for use in a computer network and comprising

a database with which the software application system may communicate so as to store structured data in and retrieve structured data from said database, and,

integrated with the software application system, a document management system comprising one or more individual components capable of interfacing with the software application system through platform-independent communication mechanisms, the interfacing enabling the software application system to control storage of documents and meta data related to the documents and retrieval of documents and meta data by the document management system,

the document management system comprising

one or more individual components residing on one or several computers and

a database with which one or more of said individual components of the document management system may communicate so as to manage documents and meta data relating to the documents, including storing documents and meta data and retrieving documents and meta data from said database, the database containing both documents and meta data pertaining to the respective documents.

2. A system according to claim 1, wherein the database of the document management system is the same database as is employed by the software application system.

3. A system according to claim 1 or 2, wherein the document management system database is a relational or an object database.

4. A system according to any of the preceding claims, wherein the document management system database is a database which is accessible through a standard communication mechanism.

5. A system according to claim 4, wherein the interfacing between the document management system database and one or more of the individual components of the document management system is through a platform-independent communication mechanism.

6. A system according to claim 5, wherein the platform-independent communication mechanism through which the database is interfaced with one or more of the individual components of the document management system is JDBC or a variant or derivative thereof.

7. A system according to any of the preceding claims, wherein individual components are implemented in a platform-independent programming language.

8. A system according to claim 7, wherein the platform-independent programming language is Java or a variant or derivative thereof

9. A system according to claim 7 or 8, wherein individual components of the software application system implemented in the platform-independent programming language are organised according to standards for creating components in that language.

10. A system according to claim 9, wherein the platform-independent programming language is Java or a variant or derivative thereof, and the standard for creating components in the language is Java Beans or a variant or derivative thereof.

11. A system according to any of the preceding claims, wherein the platform-independent communication mechanisms through which the software application system, or one or several components thereof, is interfacing with the document management system or components thereof is CORBA or a variant or derivative thereof.

12. A system according to any of the preceding claims, wherein the interaction between at least some of said individual components of the software application system is through platform-independent communication mechanisms.

13. A system according to claim 12, wherein the platform-independent communication mechanisms through which the individual components of the software application system interact is CORBA or a variant or derivative thereof.

14. A system according to any of the preceding claims, wherein the interaction between individual components of the document management system is through platform-independent communication mechanisms.

15. A system according to claim 14, wherein the platform-independent communication mechanisms through which the individual components of the document management system interact is CORBA or a variant or derivative thereof.

16. A system according to any of the preceding claims, wherein the document management system or one or more individual components thereof is/are capable of interfacing with external systems or components thereof, including the software application system or components thereof through platform-independent communication mechanisms.

17. A system according to claim 16, wherein the platform-independent communication mechanism through which the document management system or one of several components thereof is capable of interfacing with external systems or components thereof is CORBA or a variant or derivative thereof.

18. A system according to any of the preceding claims, wherein individual components of the document management system residing on individual computers connected to the network enable said computers to interface with the document management system or components thereof through platform-independent communication mechanisms, the interfacing enabling the individual computer to control storage of documents and meta data and retrieval of documents and meta data by the document management system.

19. A system according to claim 18, wherein individual components of the document management system enabling an individual computer connected to the network to

interface with the document management system can be accessed via the network and optionally downloaded onto a computer connected to the network.

20. A system according to claim 19, wherein said components are downloaded to the computer upon establishment of connection via the network between the document management system and the computer.

21. A system according to claim 20, wherein a connection between the document management system and a computer connected to the network is established with a general purpose interfacing software, such as a browser, operated by said computer.

22. A system according to any of claims 18-21, wherein the platform-independent communication mechanism through which the document management system or one of several components thereof interface with the individual computer is CORBA or a variant or derivative thereof.

23. A system according to any of the claims 18-22, wherein the individual components of the document management system enabling an individual computer connected to the network to interface with the document management system are implemented in a platform-independent programming language.

24. A system according to claim 23, wherein the platform-independent programming language is Java or a variant or derivative thereof

25. A system according to claim 23 or 24, wherein individual components implemented in the platform-independent programming language are organised according to standards for creating components in that language.

26. A system according to claim 25, wherein the platform-independent programming language is Java or a variant or derivative thereof, and the standard for creating components in the language is Java Beans or a variant or derivative thereof.

27. A system according to any of the preceding claims, wherein the software application system interfaces with the document management system via one or more

layers selected from the group consisting of CORBA IDL files, Component Object Model interfaces, Java Beans components and Active X components or variants or derivatives thereof.

28. A system according to claim 27, wherein the software application system comprises an individual component interfacing the one or more layers and the document management system by means of platform-independent communication mechanism.

29. A system according to claim 28, wherein the individual component is implemented in Java or a variant or derivative thereof.

30. A system according to claim 28 or 29, wherein the platform-independent communication mechanism is CORBA or a variant or derivative thereof.

31. A system according to any of the preceding claims, wherein the document management system comprises at least the following components:

- an index card component for creating index cards for new documents to be stored by the system and for editing existing index cards, the index carts comprising meta data pertaining the respective documents,
- an archive component for creating and updating documents in the database, each document comprises at least one index cards relating to the document data and optionally one or more sets of data, such as graphic image data or text data, and
- a search/retrieve component for searching the database for documents according to a set of search criteria and producing a set of data representing the result of the search, and for retrieving documents from the database.

32. A system according to claim 31, wherein the document management system additionally comprises one or several of the following components:

- a printing component for printing sets of data, such as graphic image data or text data, that have been received by the system or that have been retrieved from the database,

- a security component for imposing regulations on the accessibility of documents in the database for the users of the system,
- a viewing component for producing a representation on a graphical display of sets of data, such as graphic image data or text data, that have been received by the system from external systems or components, or that have been retrieved from the database, and
- a directory component for producing a representation on a graphical display of the content of a file storage means, allowing a user to select a file from the file storage means and adding said file to a document in the database.

33. A system according to any of the preceding claims, wherein the document management system further comprises

- at least one peripheral means capable of producing an output comprising data representing a physical phenomenon external to the system, the at least one peripheral means and at least one of said computers being interconnected, optionally on the network, and the control of the operation of the at least one peripheral means, including handling of output and storage of at least a part of the output as at least a part of a document in the database, being performed by one or more of said individual components.

34. A system according to claim 33, wherein at least one of the at least one peripheral means is a device generating an output representing digitized graphical image data, such as a scanner for scanning paper documents and producing an output file comprising graphic image data representing the paper document.

35. A system according to claim 33 or 34, wherein the interfacing between the at least one peripheral means and the components that control the operation of the peripheral means is being through platform-independent communication mechanisms.

36. A system according to any of claims 33-35, wherein at least one of the at least one peripheral means is a device generating an output representing digitized graphical image data, such as a scanner and the platform-independent communication

mechanisms through which the at least one scanner and the components that control the operation of the device is TWAIN or a variant or derivative thereof.

37. A system according to any of claims 33-35, wherein at least one of the at least one peripheral means is a scanner and the platform-independent communication mechanisms through which the at least one scanner and the components that control the operation of the scanner is ISIS or a variant or derivative thereof.

38. A system according to any of claims 33-37, wherein at least one of the at least one peripheral means is capable of producing an output comprising data representing sound.

39. A system according to any of claims 33-38, wherein the document management system comprises a controlling component for controlling the operation of the peripheral means and handling the output from the peripheral means.

40. A system according to any of the preceding claims, wherein the document management system comprises an optical character recognition module for analysing the content of a set of graphic image data representing a document containing textual matter and producing an output file comprising text data that represent at least a part of said textual matter, the control of the operation of the optical character recognition module, including handling of output, being performed by one or more of said individual components.

41. A system according to any of the preceding claims, wherein at least one of said computers comprises a graphical display and input means, such as a keyboard and optionally a pointing device, such as a mouse, the communication of information from the system to a user of the system being performed with the use of a graphical user interface displayed on the graphical display, and the communication of information from the user to the system being performed with the input means.

42. A system according to any of the preceding claims, wherein said software application system is a line of business application system.

43. A system according to claim 43, wherein said line of business application system is selected from a manufacturing system, an accounting system, a Custom Relationship Management (CRM) system, an Enterprise Resource Planning (ERP) system, a health care system and a financial application.

44. A document management system suitable for use in establishing a software application system with integrated document management system as claimed in any of claims 1-43, the document management system comprising:

one or more individual components residing on one or several computers, and being accessible from one or more computers connected to the network, and

a database with which one or more of said individual components of the document management system may communicate so as to manage documents and meta data pertaining to the documents, including storing documents with meta data relating to the documents in the database and retrieving documents and meta data from the database, the database containing both documents and meta data pertaining to the respective documents, and

integration layers enabling integration between the document management system and the software application system in such a manner that the document management system can be controlled from within the software application.

45. A document management system according to claim 44, wherein the interaction layers are selected from the group consisting of CORBA IDL files, COM interfaces, Java Beans components and Active X components or variants or derivatives thereof.

46. A document management system according to claim 44 or 45, which is implemented in Java or a variant or derivative thereof.

47. A document management system according to any of claims 44-46, which has the features defined for the document management system in any of claims 1-43.

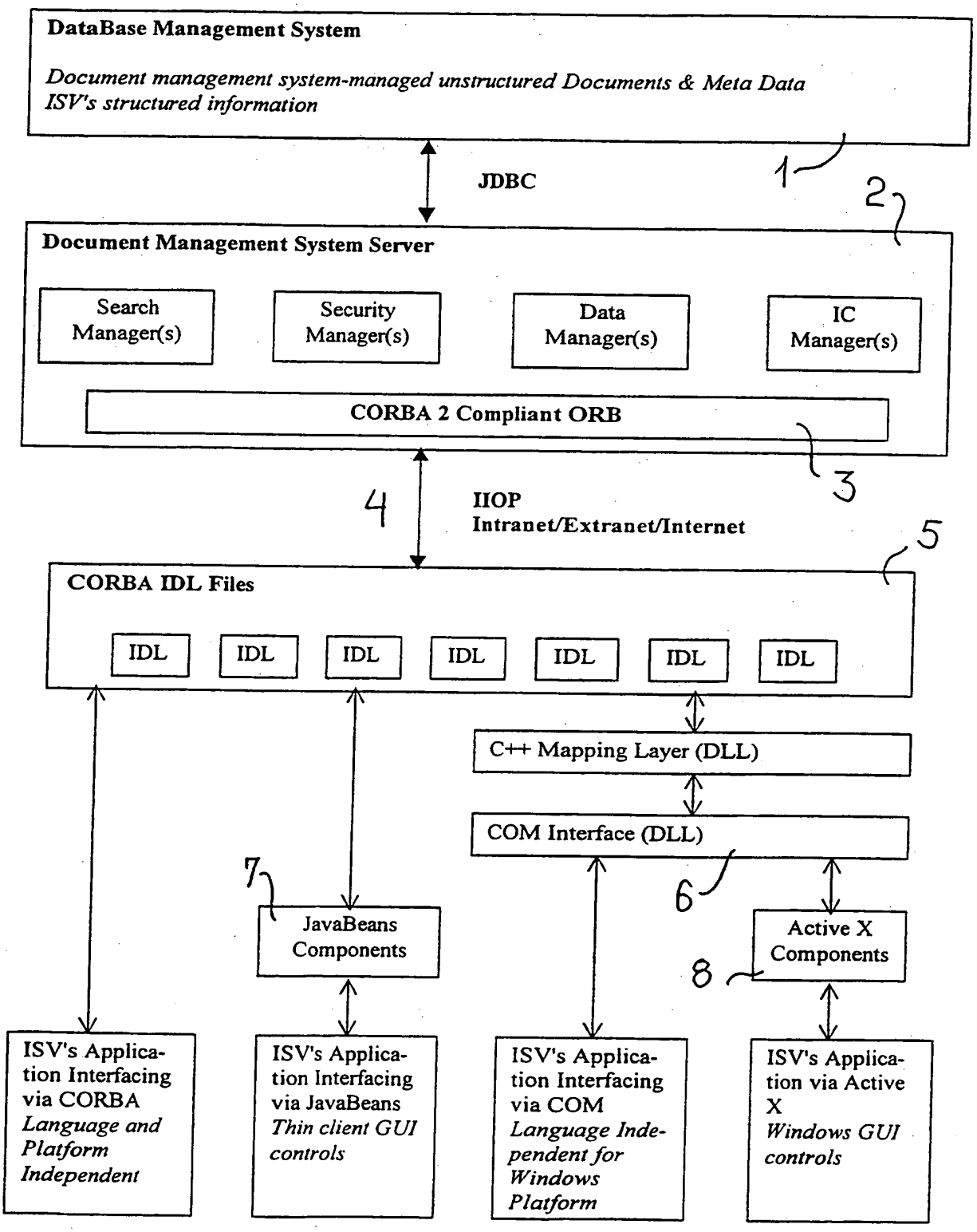

DataBase Management System
Document management system-managed unstructured Documents & Meta Data
ISV's structured information
JDBC
1
2
Document Management System Server
Search Manager(s)
Security Manager(s)
Data Manager(s)
IC Manager(s)
CORBA 2 Compliant ORB
3
4
IIOP
Intranet/Extranet/Internet
5
CORBA IDL Files
IDL
IDL
IDL
IDL
IDL
IDL
IDL
C++ Mapping Layer (DLL)
COM Interface (DLL)
6
7
JavaBeans Components
Active X Components
8
ISV's Application Interfacing via CORBA
Language and Platform Independent
ISV's Application Interfacing via JavaBeans
Thin client GUI controls
ISV's Application Interfacing via COM
Language Independent for Windows Platform
ISV's Application via Active X
Windows GUI controls

Fig. 1

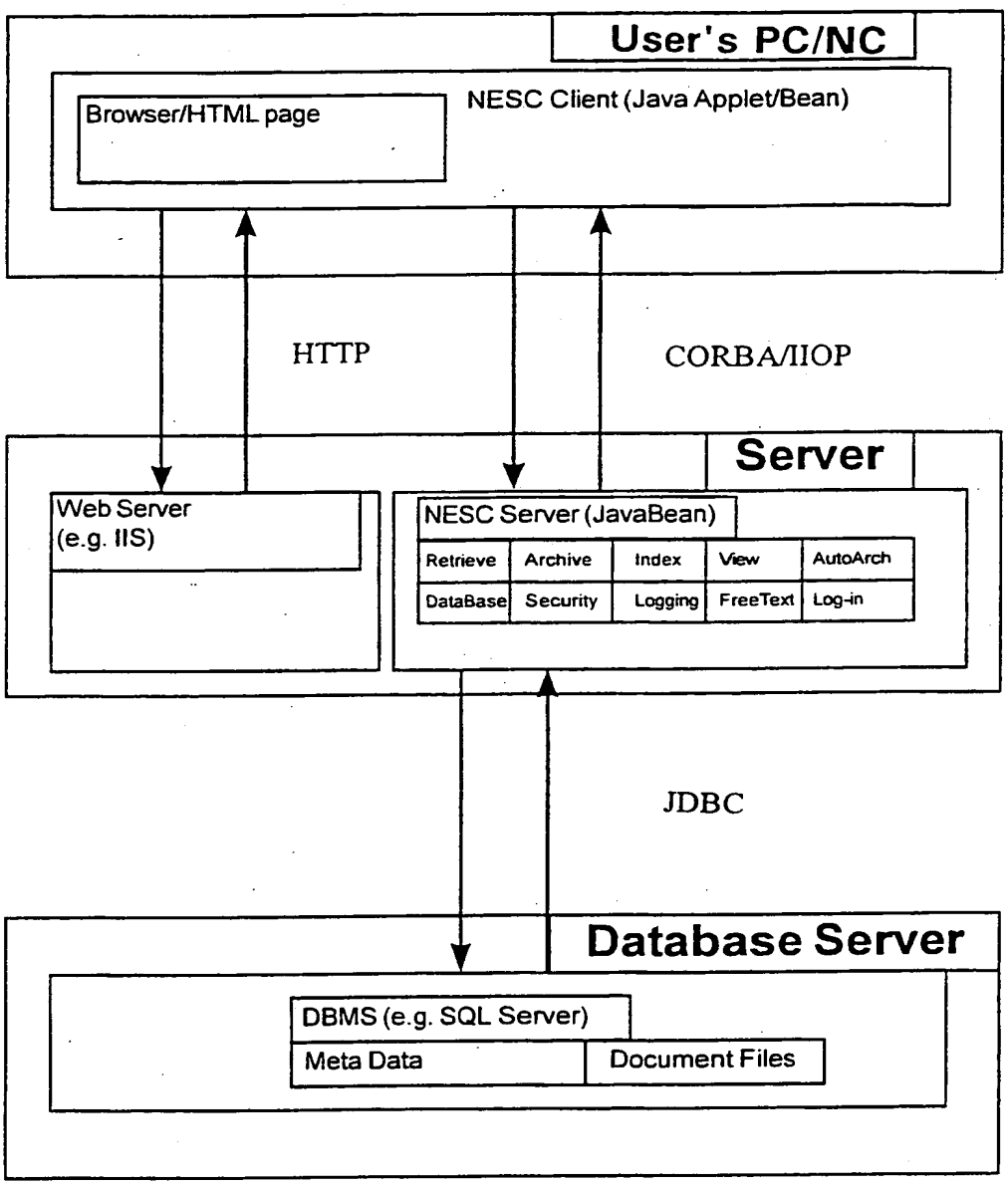

User's PC/NC
Browser/HTML page
NESC Client (Java Applet/Bean)
HTTP
CORBA/IIOP
Server
Web Server
(e.g. IIS)
NESC Server (JavaBean)
Retrieve
Archive
Index
View
AutoArch
DataBase
Security
Logging
FreeText
Log-in
JDBC
Database Server
DBMS (e.g. SQL Server)
Meta Data
Document Files

Fig. 2

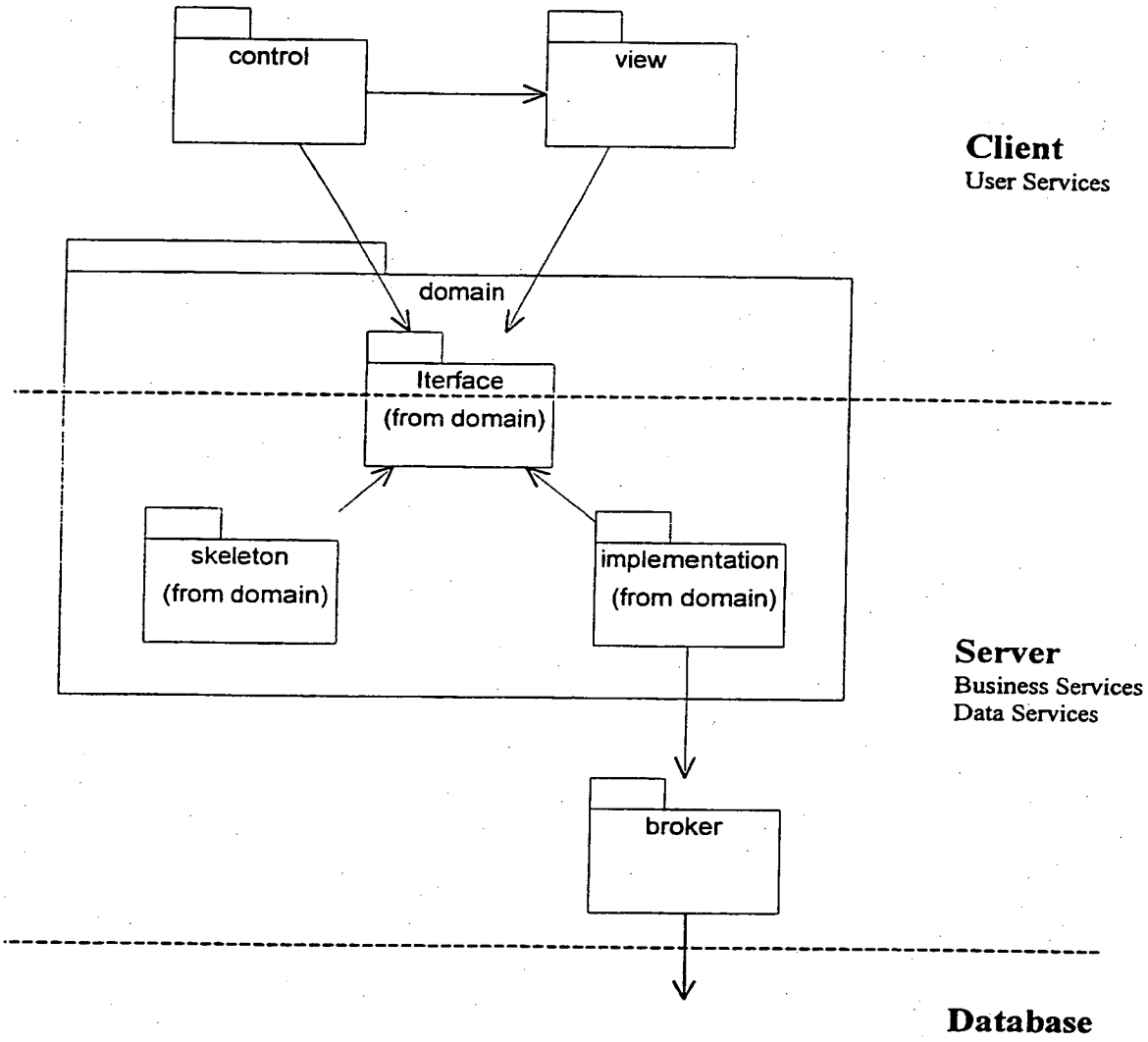

control
view
Client
User Services
domain
Iterface
(from domain)
skeleton
(from domain)
implementation
(from domain)
Server
Business Services
Data Services
broker
Database

Fig. 3

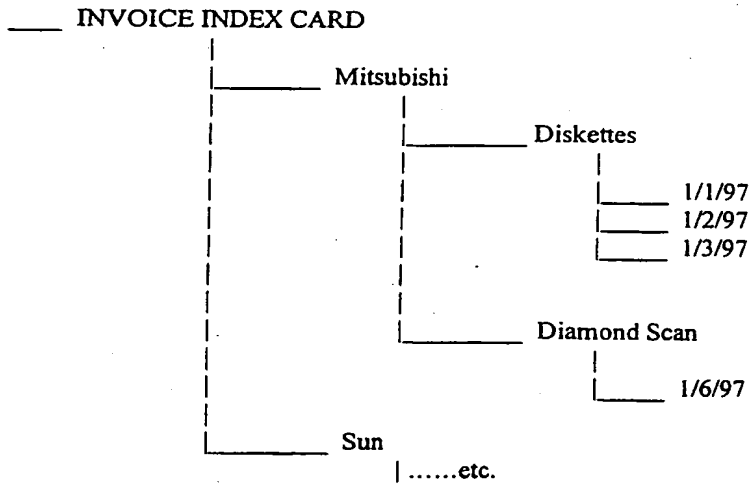

INVOICE INDEX CARD
Mitsubishi
Diskettes
1/1/97
1/2/97
1/3/97
Diamond Scan
1/6/97
Sun
.....etc.

Fig. 4

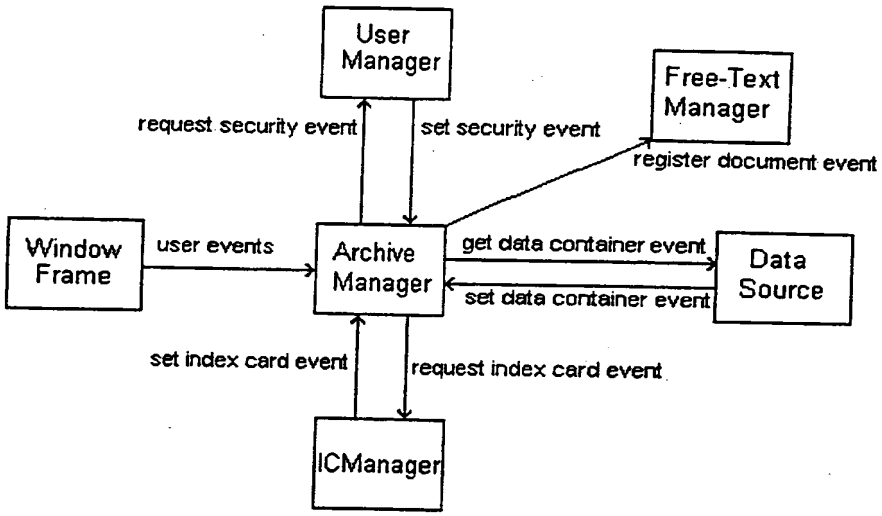

User Manager
Free-Text Manager
request security event
set security event
register document event
Window Frame
user events
Archive Manager
get data container event
set data container event
Data Source
set index card event
request index card event
ICManager

Fig. 5

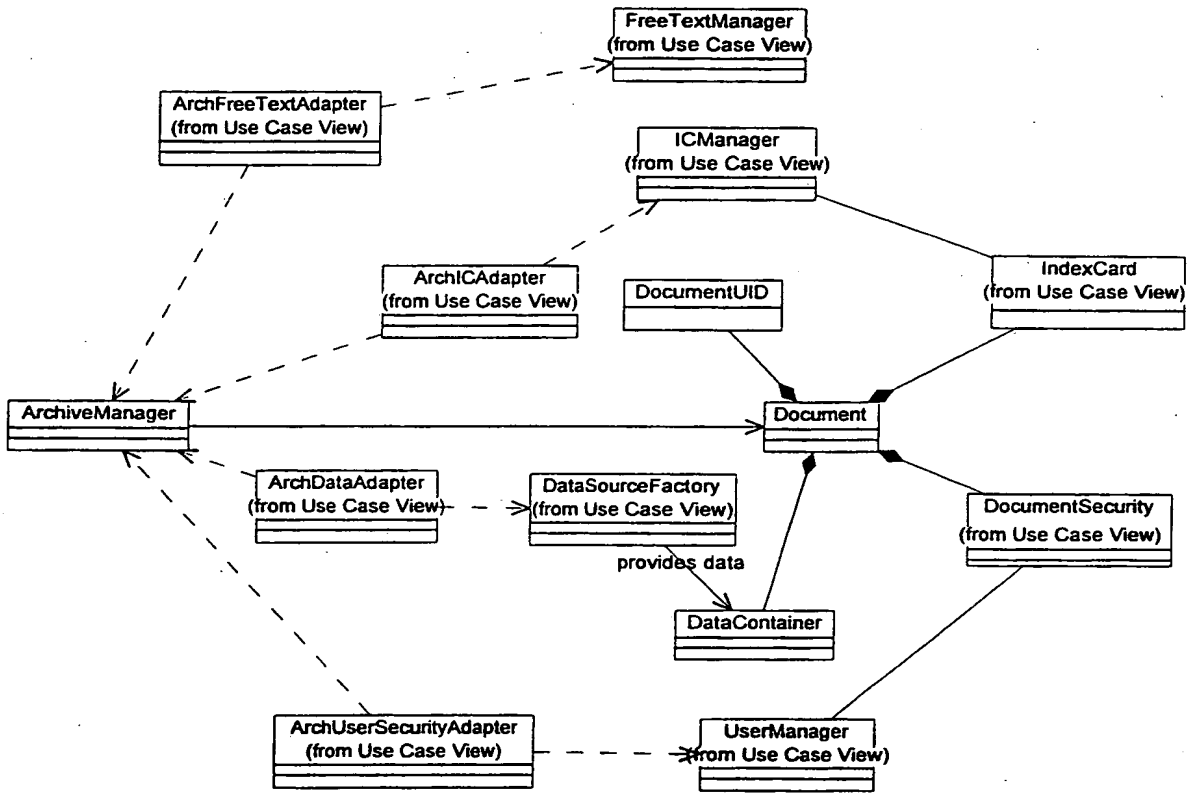

FreeTextManager
(from Use Case View)
ArchFreeTextAdapter
(from Use Case View)
ICManager
(from Use Case View)
ArchICAdapter
(from Use Case View)
DocumentUID
IndexCard
(from Use Case View)
ArchiveManager
Document
ArchDataAdapter
(from Use Case View)
DataSourceFactory
(from Use Case View)
DocumentSecurity
(from Use Case View)
provides data
DataContainer
ArchUserSecurityAdapter
(from Use Case View)
UserManager
(from Use Case View)

Fig. 6

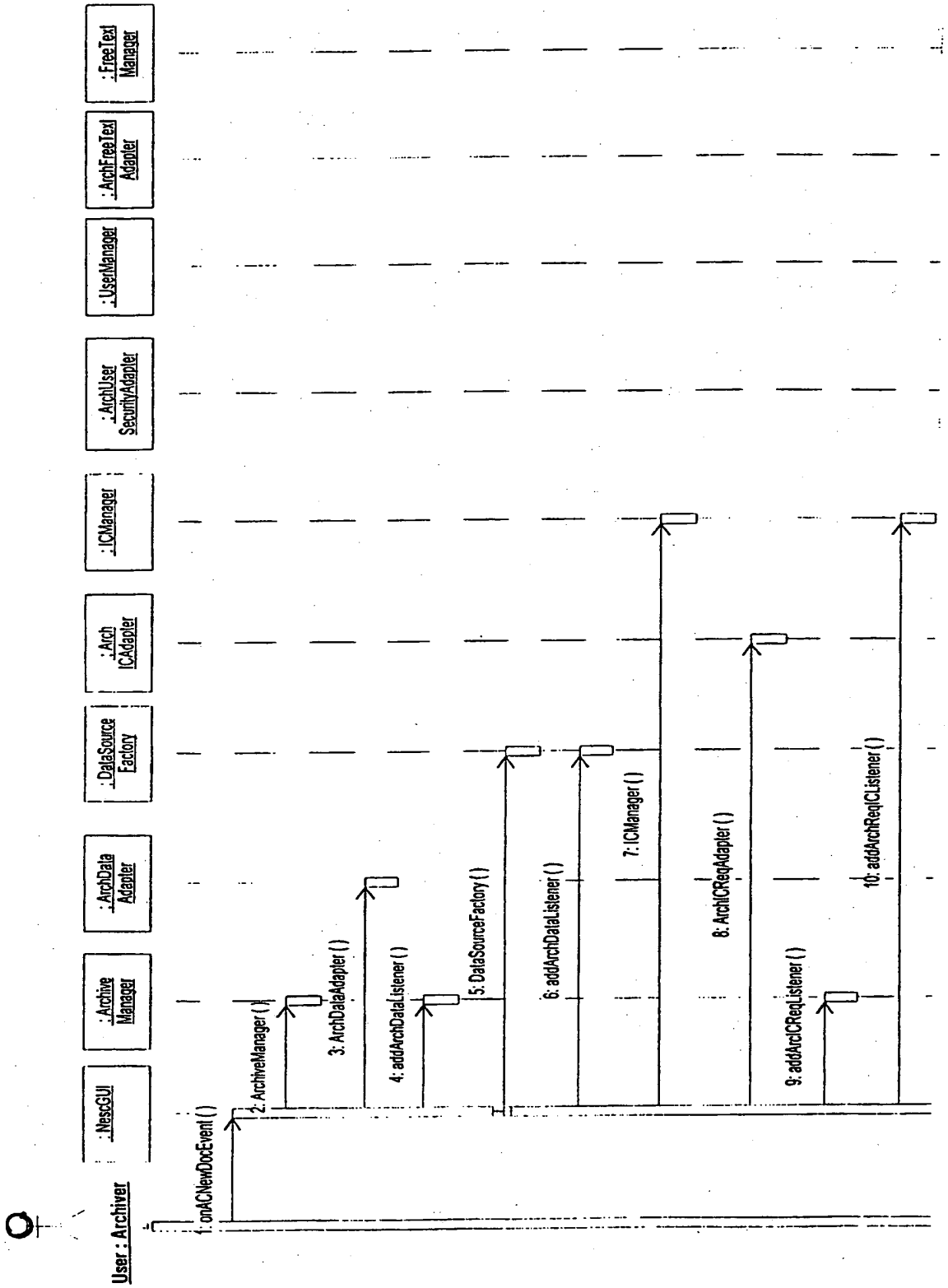

User : Archiver
: NescGUI
: Archive Manager
: ArchData Adapter
: DataSource Factory
: Arch ICAdapter
: ICManager
: ArchUser SecurityAdapter
: UserManager
: ArchFreeText Adapter
: FreeText Manager
1: onACNewDocEvent ( )
2: ArchiveManager ( )
3: ArchDataAdapter ( )
4: addArchDataListener ( )
5: DataSourceFactory ( )
6: addArchDataListener ( )
7: ICManager ( )
8: ArchICReqAdapter ( )
9: addArcICReqListener ( )
10: addArchReqICListener ( )

Fig. 7a

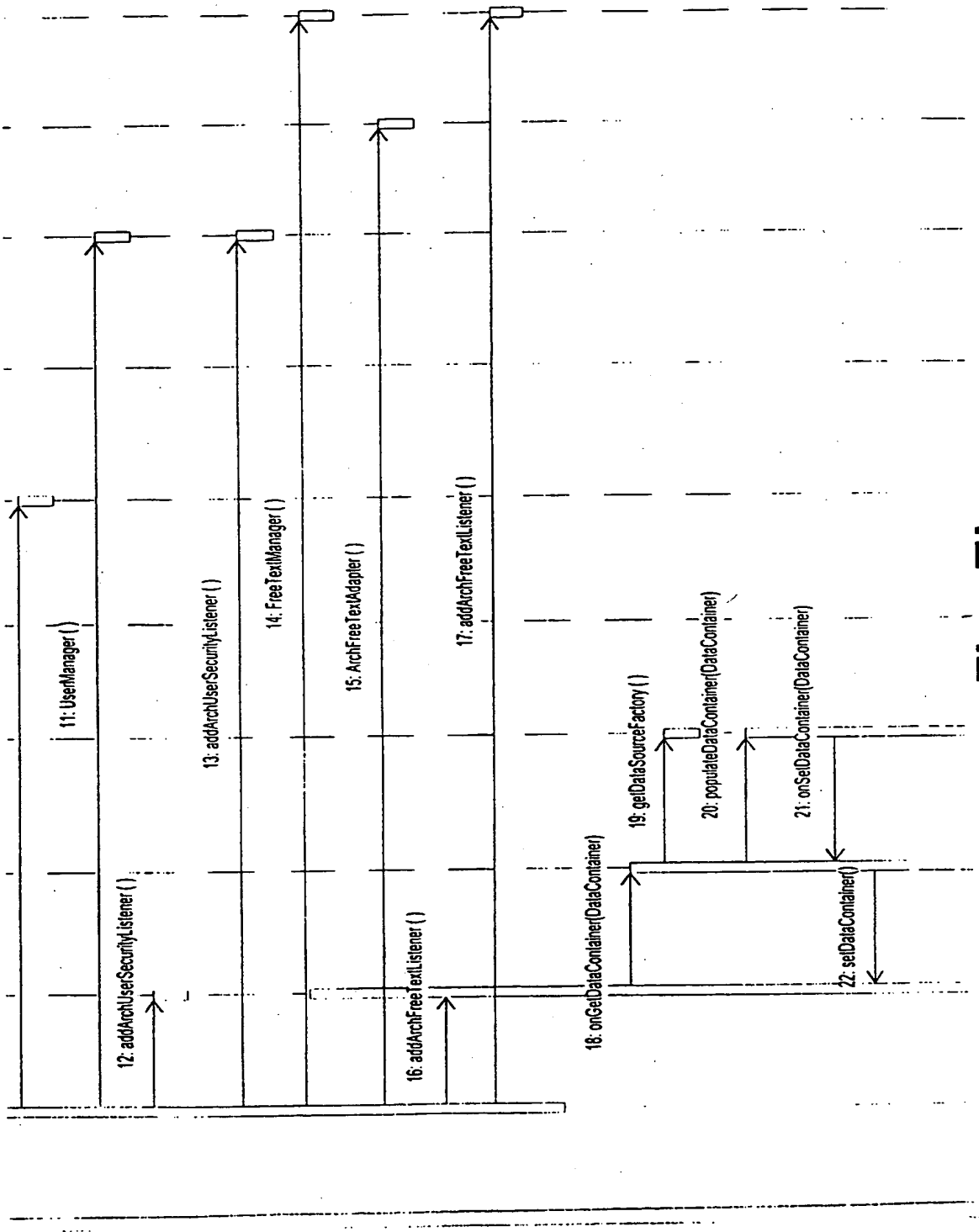

11: UserManager ( )
12: addArchUserSecurityListener ( )
13: addArchUserSecurityListener ( )
14: FreeTextManager ( )
15: ArchFreeTextAdapter ( )
16: addArchFreeTextListener ( )
17: addArchFreeTextListener ( )
18: onGetDataContainer(DataContainer)
19: getDataSourceFactory ( )
20: populateDataContainer(DataContainer)
21: onSetDataContainer(DataContainer)
22: setDataContainer( )

Fig. 7b

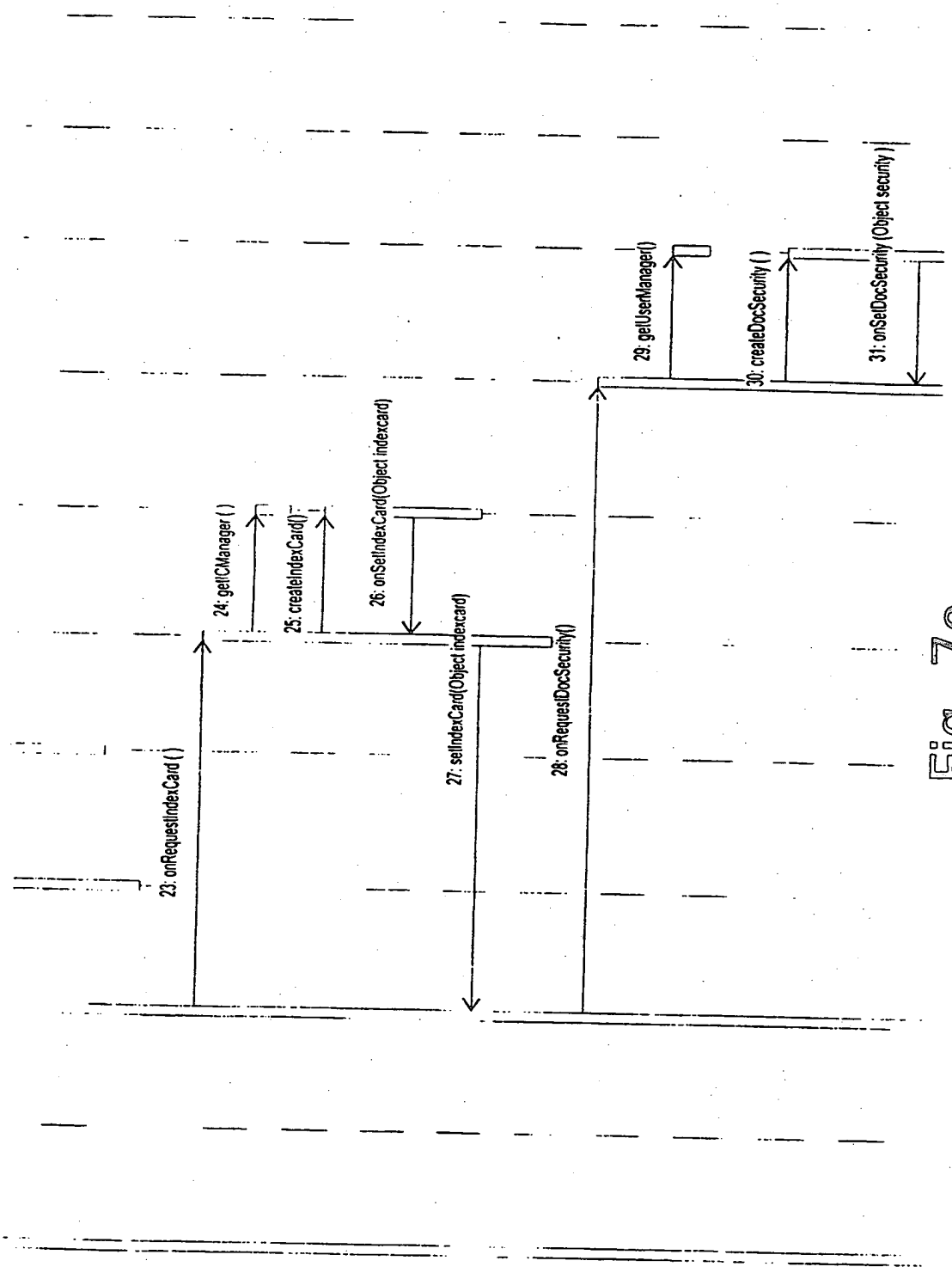

23: onRequestIndexCard ( )
24: getICManager ( )
25: createIndexCard()
26: onSetIndexCard(Object indexcard)
27: setIndexCard(Object indexcard)
28: onRequestDocSecurity()
29: getUserManager()
30: createDocSecurity ( )
31: onSetDocSecurity (Object security)

Fig. 7c

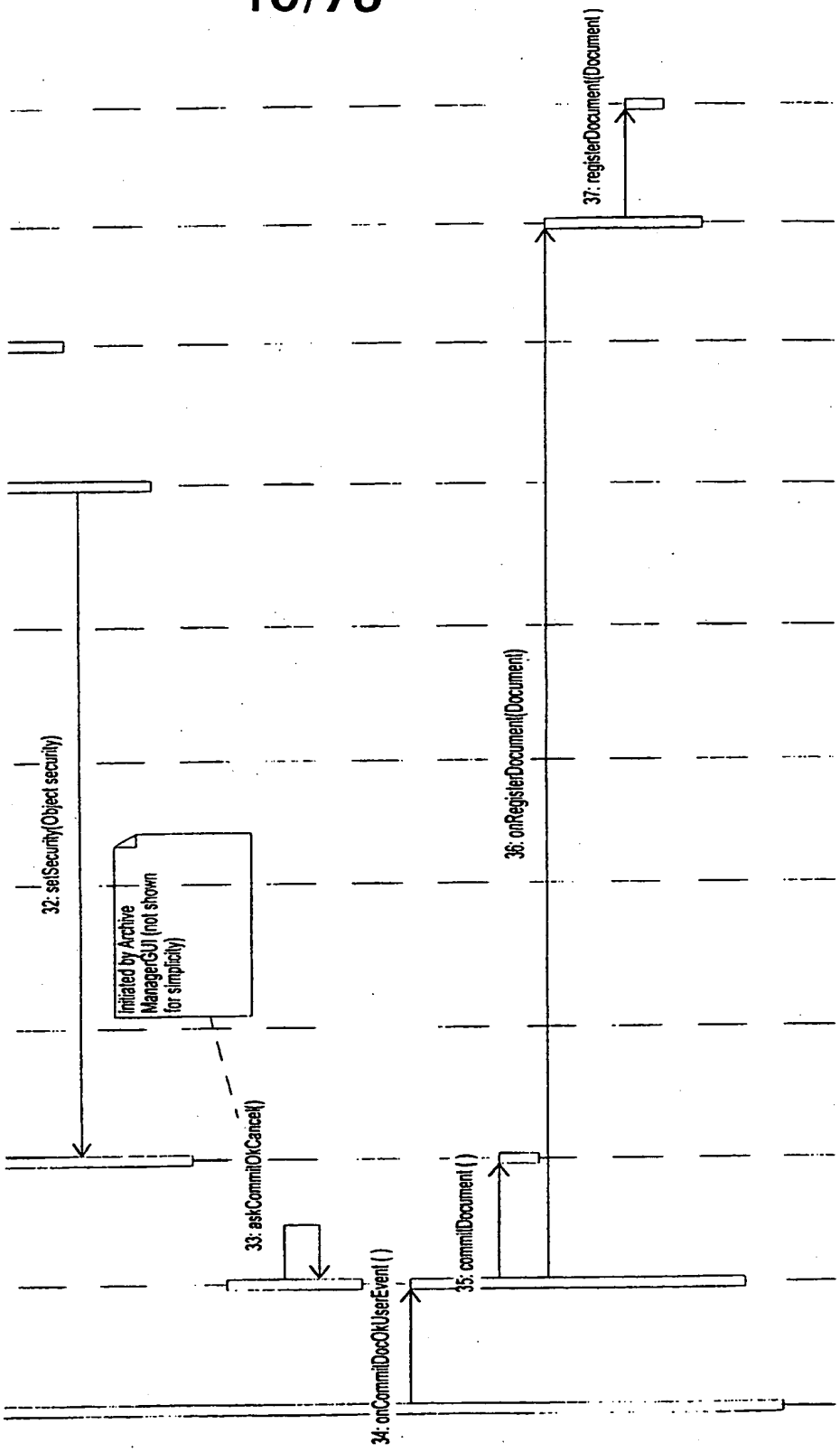

37: registerDocument(Document)
36: onRegisterDocument(Document)
32: setSecurity(Object security)
Initiated by Archive ManagerGUI (not shown for simplicity)
33: askCommitOkCancel()
35: commitDocument()
34: onCommitDocOkUserEvent()

Fig. 7d

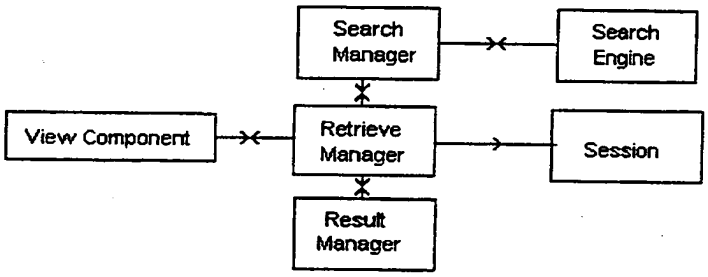

Search Manager
Search Engine
View Component
Retrieve Manager
Session
Result Manager

Fig. 8

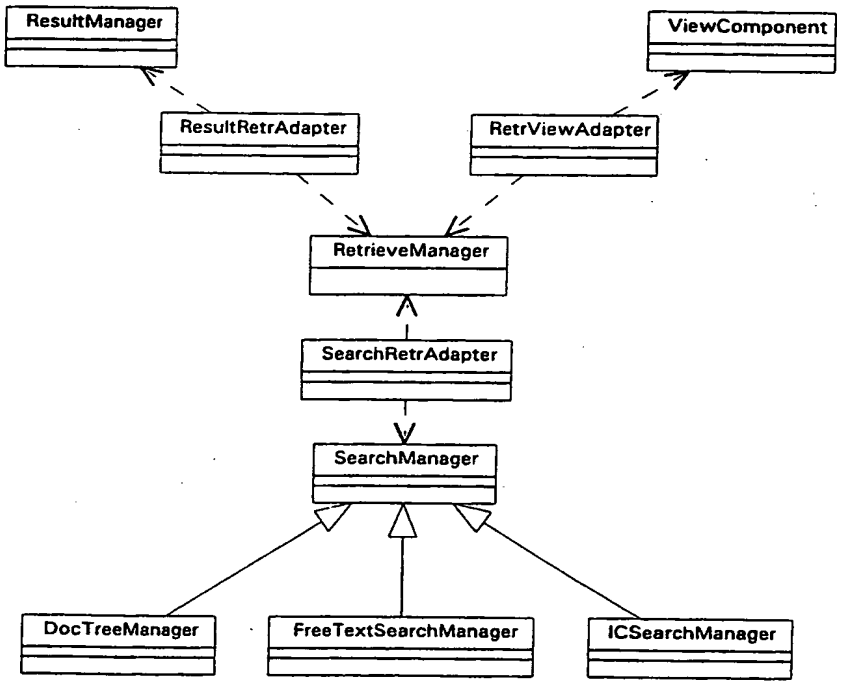

ResultManager
ViewComponent
ResultRetrAdapter
RetrViewAdapter
RetrieveManager
SearchRetrAdapter
SearchManager
DocTreeManager
FreeTextSearchManager
ICSearchManager

Fig. 9

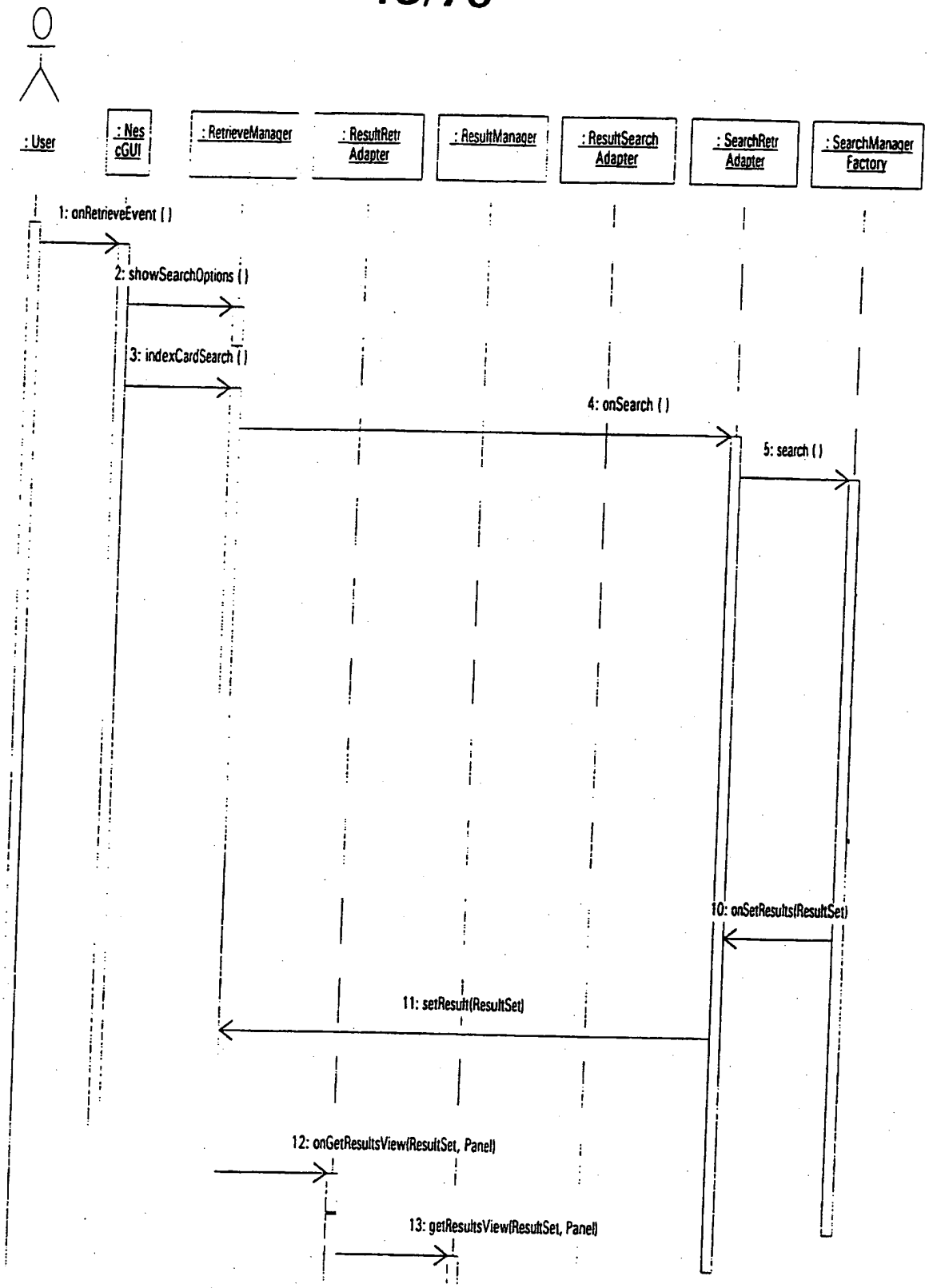

: User
: Nes cGUI
: RetrieveManager
: ResultRetr Adapter
: ResultManager
: ResultSearch Adapter
: SearchRetr Adapter
: SearchManager Factory
1: onRetrieveEvent ( )
2: showSearchOptions ( )
3: indexCardSearch ( )
4: onSearch ( )
5: search ( )
10: onSetResults(ResultSet)
11: setResult(ResultSet)
12: onGetResultsView(ResultSet, Panel)
13: getResultsView(ResultSet, Panel)

Fig. 10a

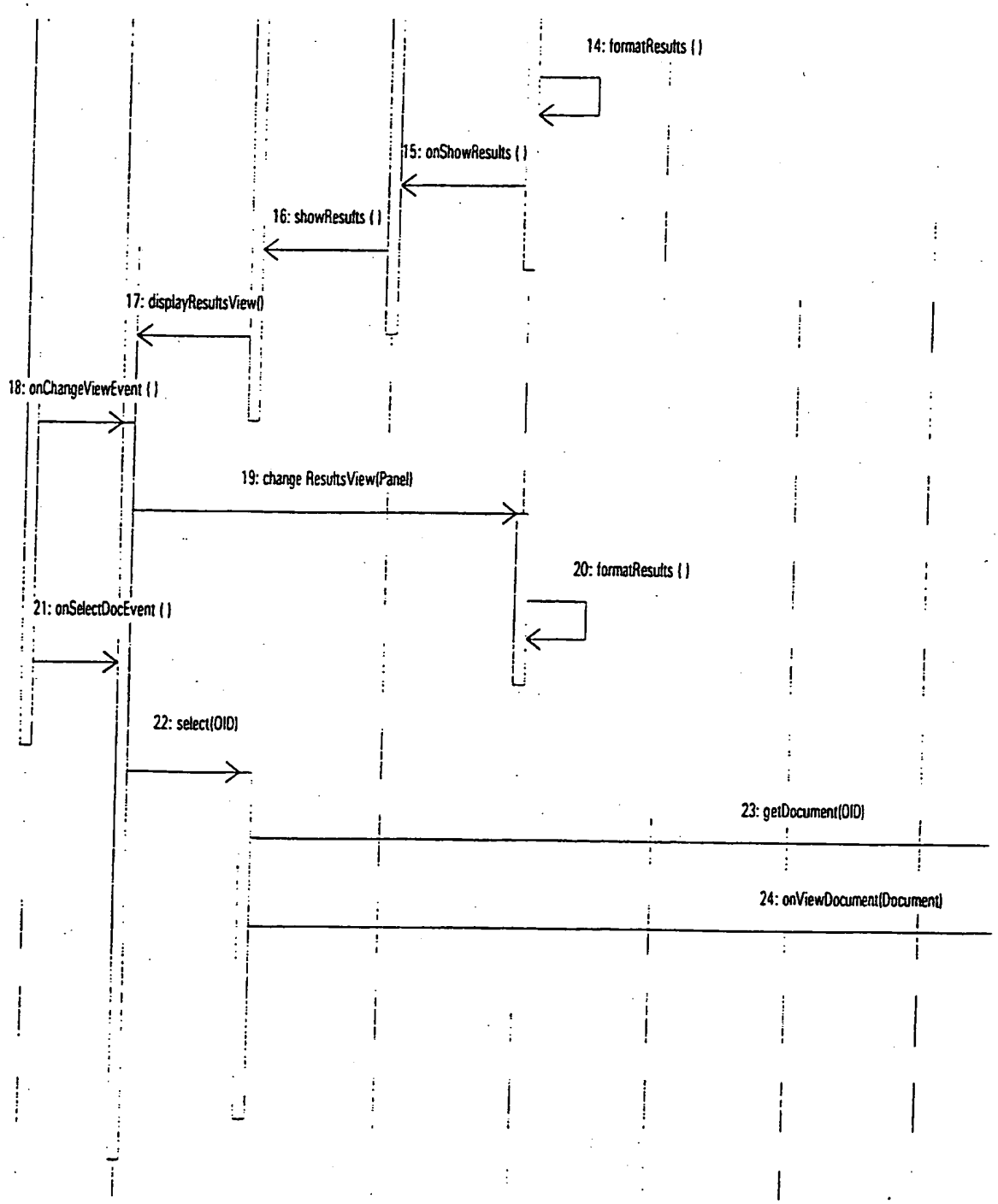

14: formatResults ( )
15: onShowResults ( )
16: showResults ( )
17: displayResultsView()
18: onChangeViewEvent ( )
19: change ResultsView(Panel)
20: formatResults ( )
21: onSelectDocEvent ( )
22: select(OID)
23: getDocument(OID)
24: onViewDocument(Document)

Fig. 10b

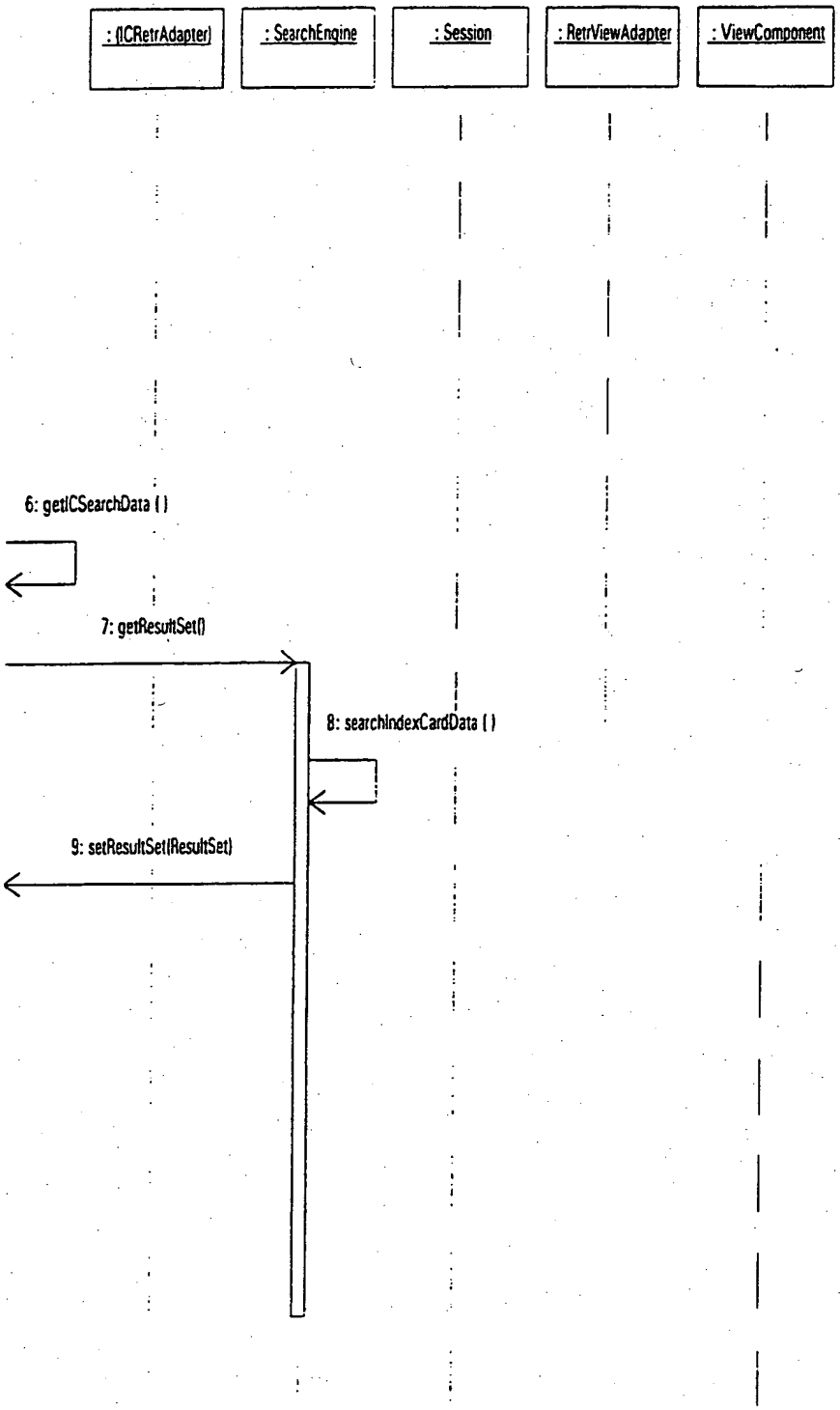

: ICRetrAdapter
: SearchEngine
: Session
: RetrViewAdapter
: ViewComponent
6: getICSearchData ( )
7: getResultSet()
8: searchIndexCardData ( )
9: setResultSet(ResultSet)

Fig. 10c

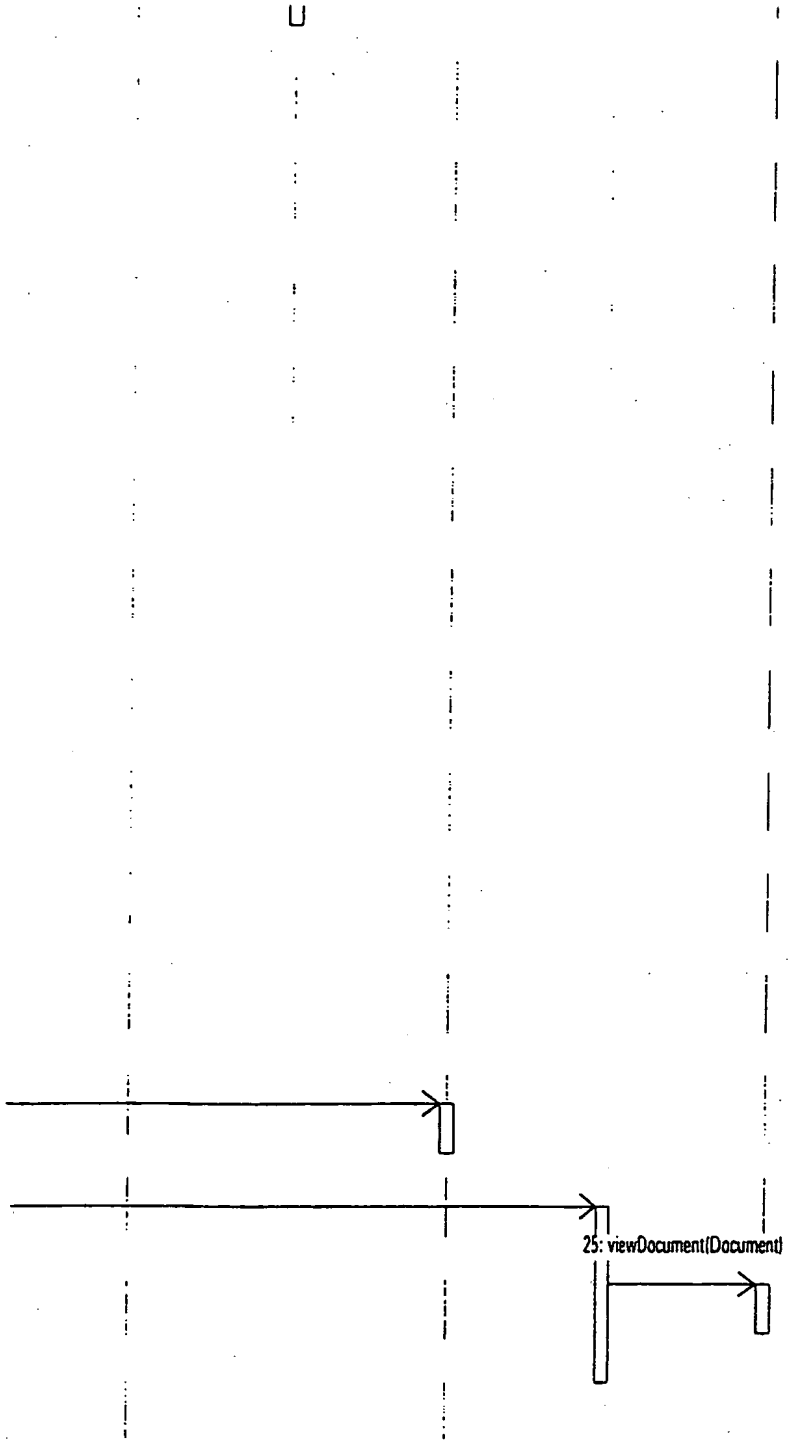


Fig. 10d

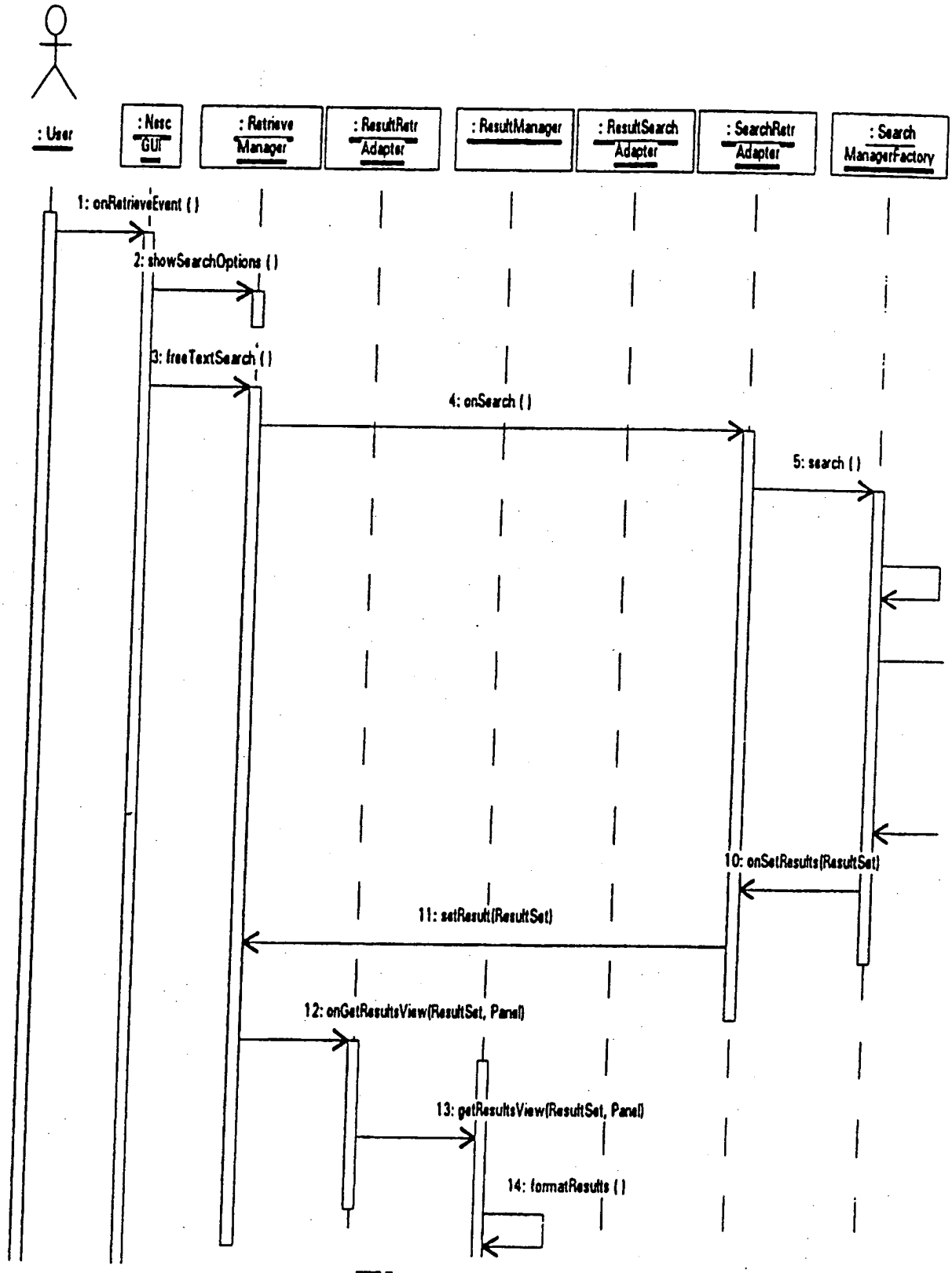

: User
: Nesc GUI
: Retrieve Manager
: ResultRetr Adapter
: ResultManager
: ResultSearch Adapter
: SearchRetr Adapter
: Search ManagerFactory
1: onRetrieveEvent ( )
2: showSearchOptions ( )
3: freeTextSearch ( )
4: onSearch ( )
5: search ( )
10: onSetResults(ResultSet)
11: setResult(ResultSet)
12: onGetResultsView(ResultSet, Panel)
13: getResultsView(ResultSet, Panel)
14: formatResults ( )

Fig. 11a

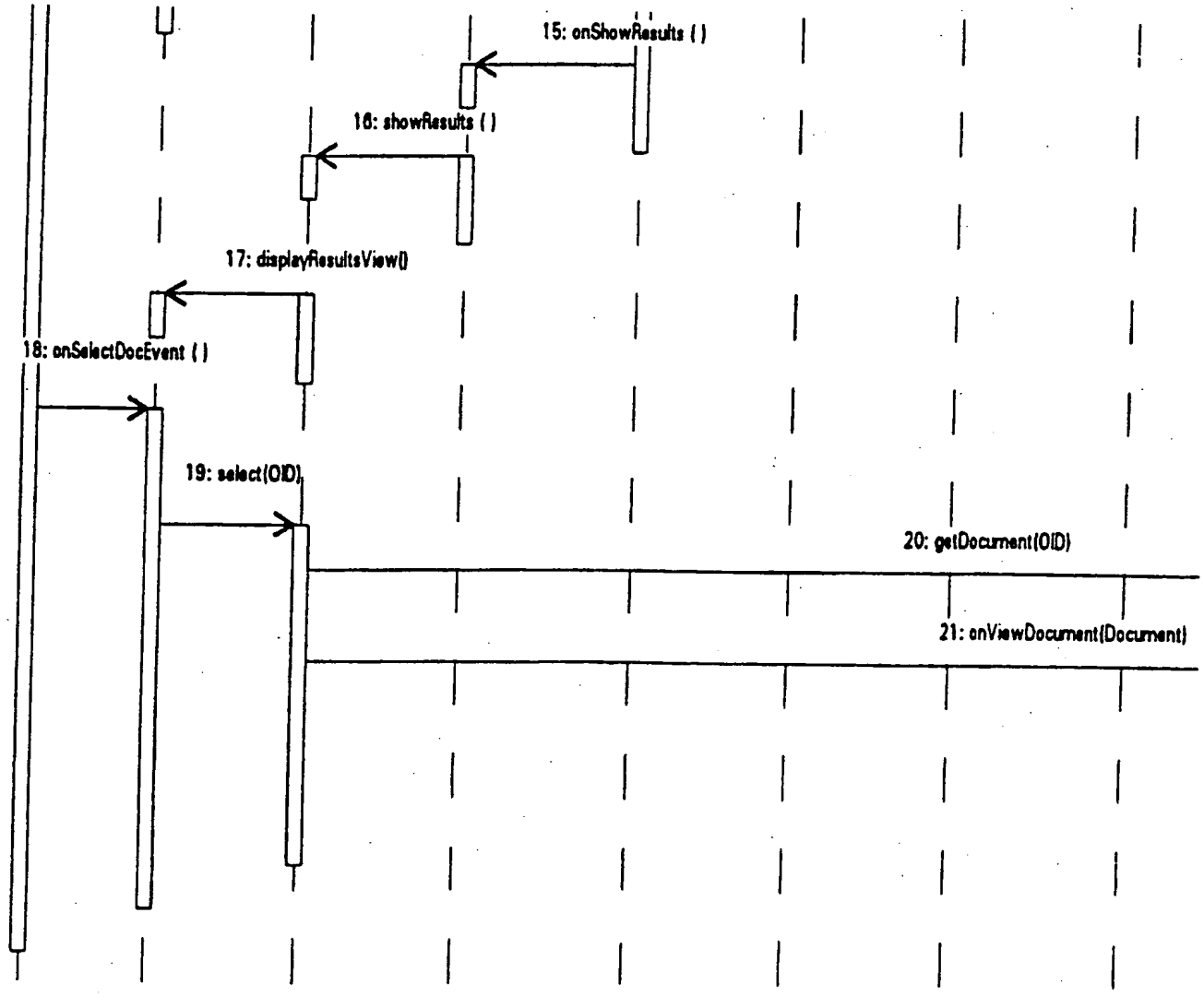

15: onShowResults ( )
16: showResults ( )
17: displayResultsView()
18: onSelectDocEvent ( )
19: select(OID)
20: getDocument(OID)
21: onViewDocument(Document)

Fig. 11b

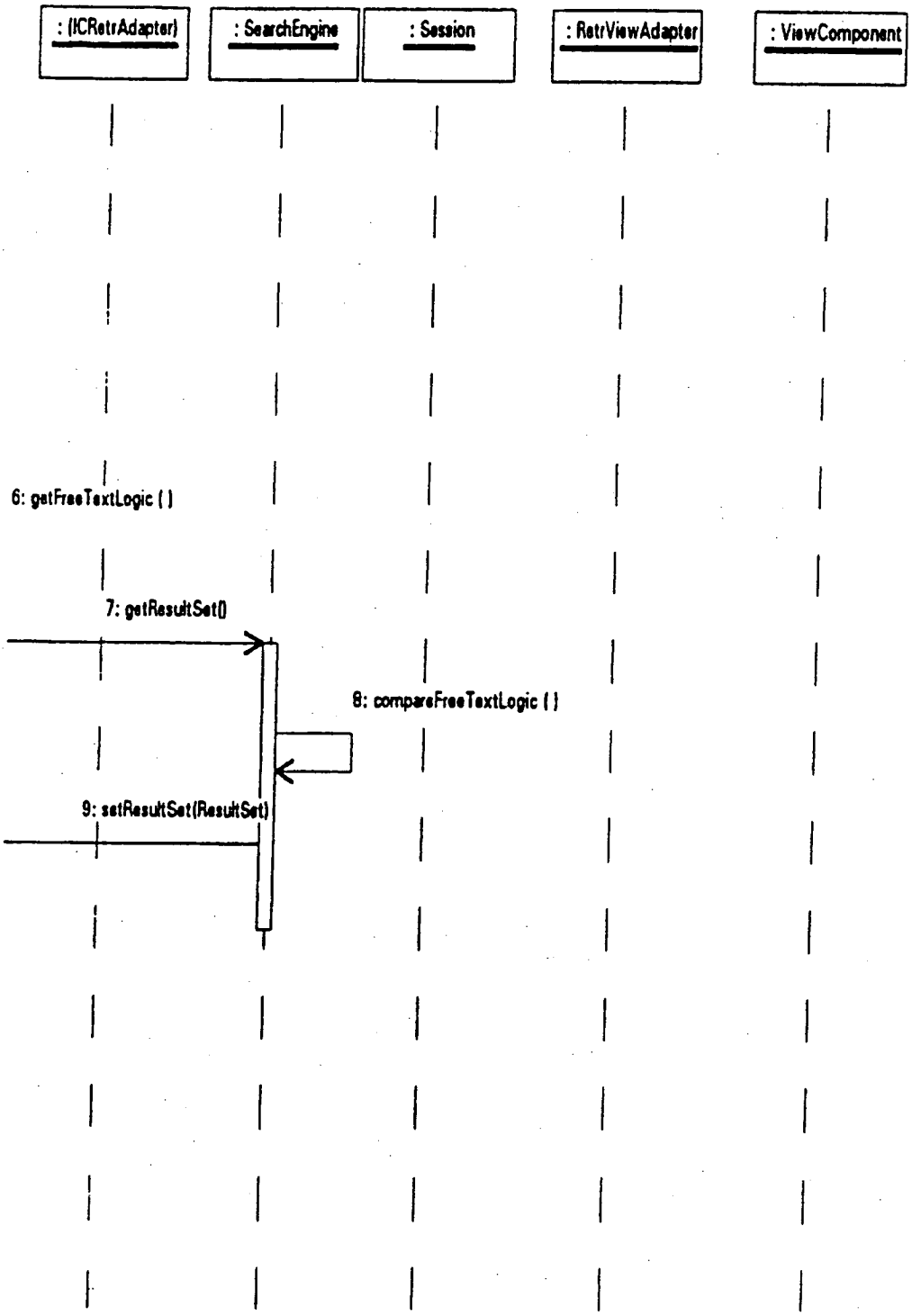

: IICRetrAdapter)
: SearchEngine
: Session
: RetrViewAdapter
: ViewComponent
6: getFreeTextLogic ( )
7: getResultSet()
8: compareFreeTextLogic ( )
9: setResultSet(ResultSet)

Fig. 11c

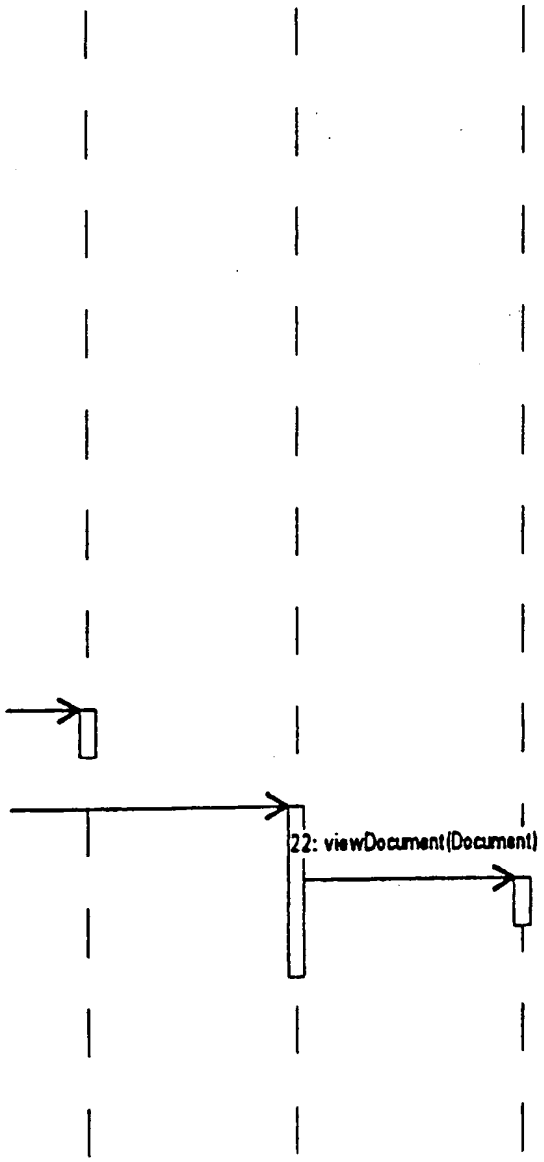


Fig. 20

Fig. 11d

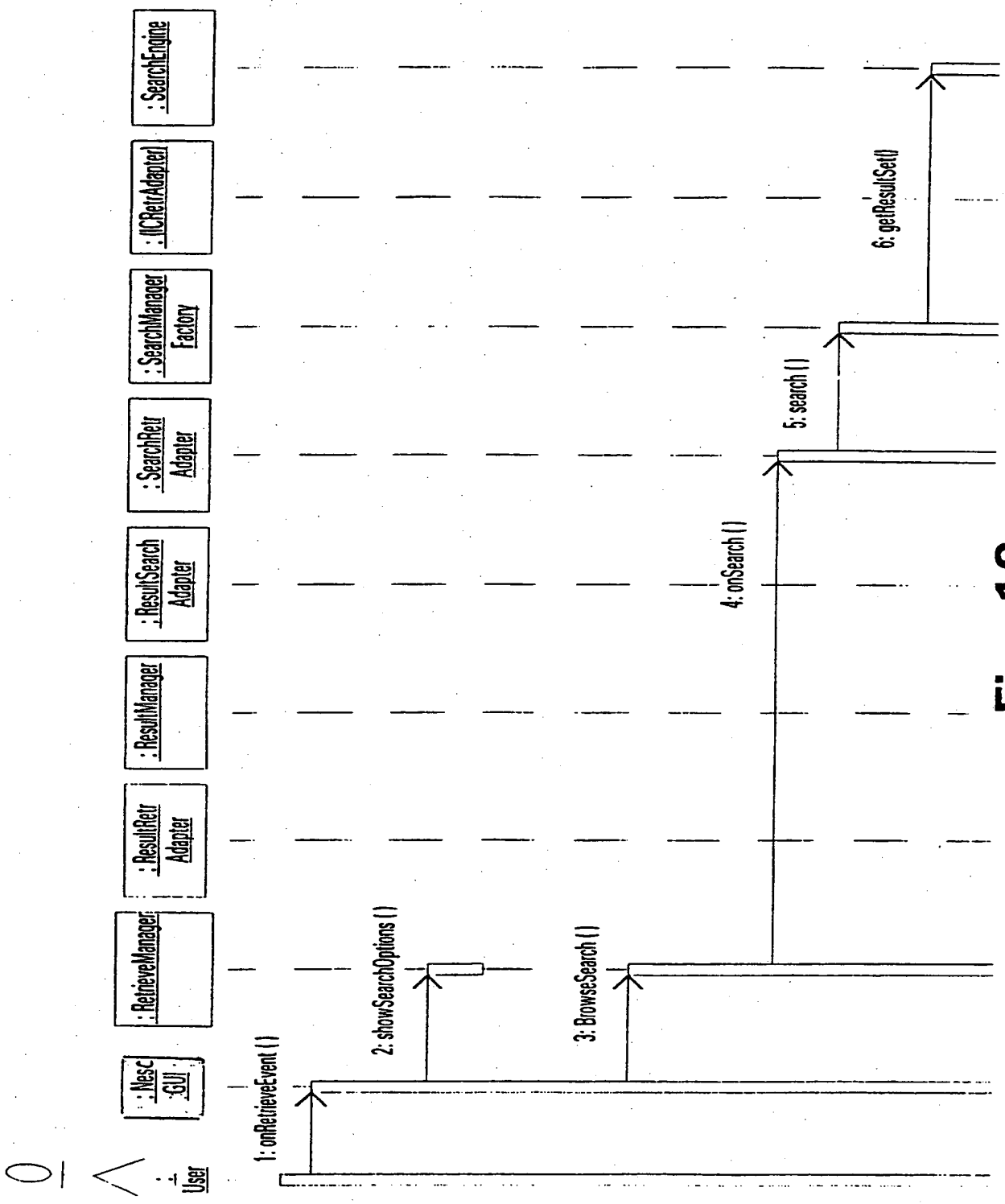

User
: NesC GUI
: RetrieveManager
: ResultRetr Adapter
: ResultManager
: ResultSearch Adapter
: SearchRetr Adapter
: SearchManager Factory
: IICRetrAdapter
: SearchEngine
1: onRetrieveEvent ( )
2: showSearchOptions ( )
3: BrowseSearch ( )
4: onSearch ( )
5: search ( )
6: getResultSet( )

Fig. 12a

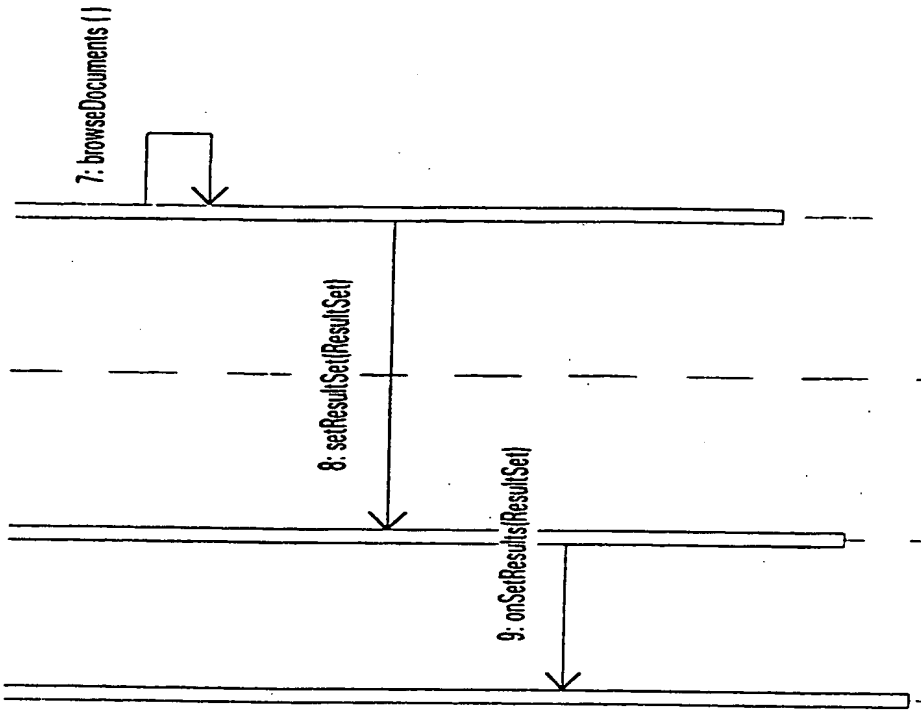

7: browseDocuments ( )
8: setResultSet(ResultSet)
9: onSetResults(ResultSet)

Fig. 12b

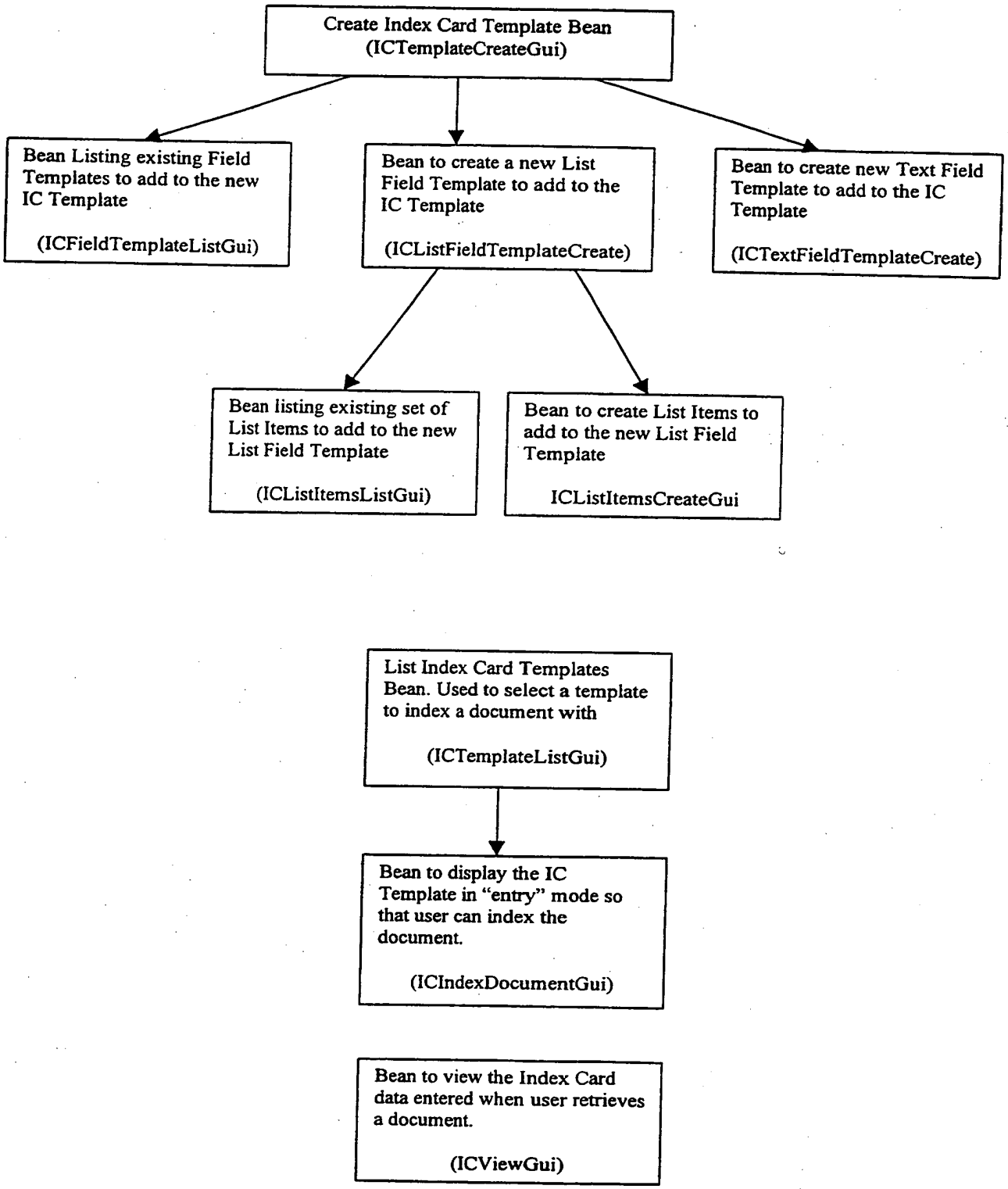

Create Index Card Template Bean (ICTemplateCreateGui)
Bean Listing existing Field Templates to add to the new IC Template (ICFieldTemplateListGui)
Bean to create a new List Field Template to add to the IC Template (ICListFieldTemplateCreate)
Bean to create new Text Field Template to add to the IC Template (ICTextFieldTemplateCreate)
Bean listing existing set of List Items to add to the new List Field Template (ICListItemsListGui)
Bean to create List Items to add to the new List Field Template ICListItemsCreateGui
List Index Card Templates Bean. Used to select a template to index a document with (ICTemplateListGui)
Bean to display the IC Template in "entry" mode so that user can index the document. (ICIndexDocumentGui)
Bean to view the Index Card data entered when user retrieves a document. (ICViewGui)

Fig. 13

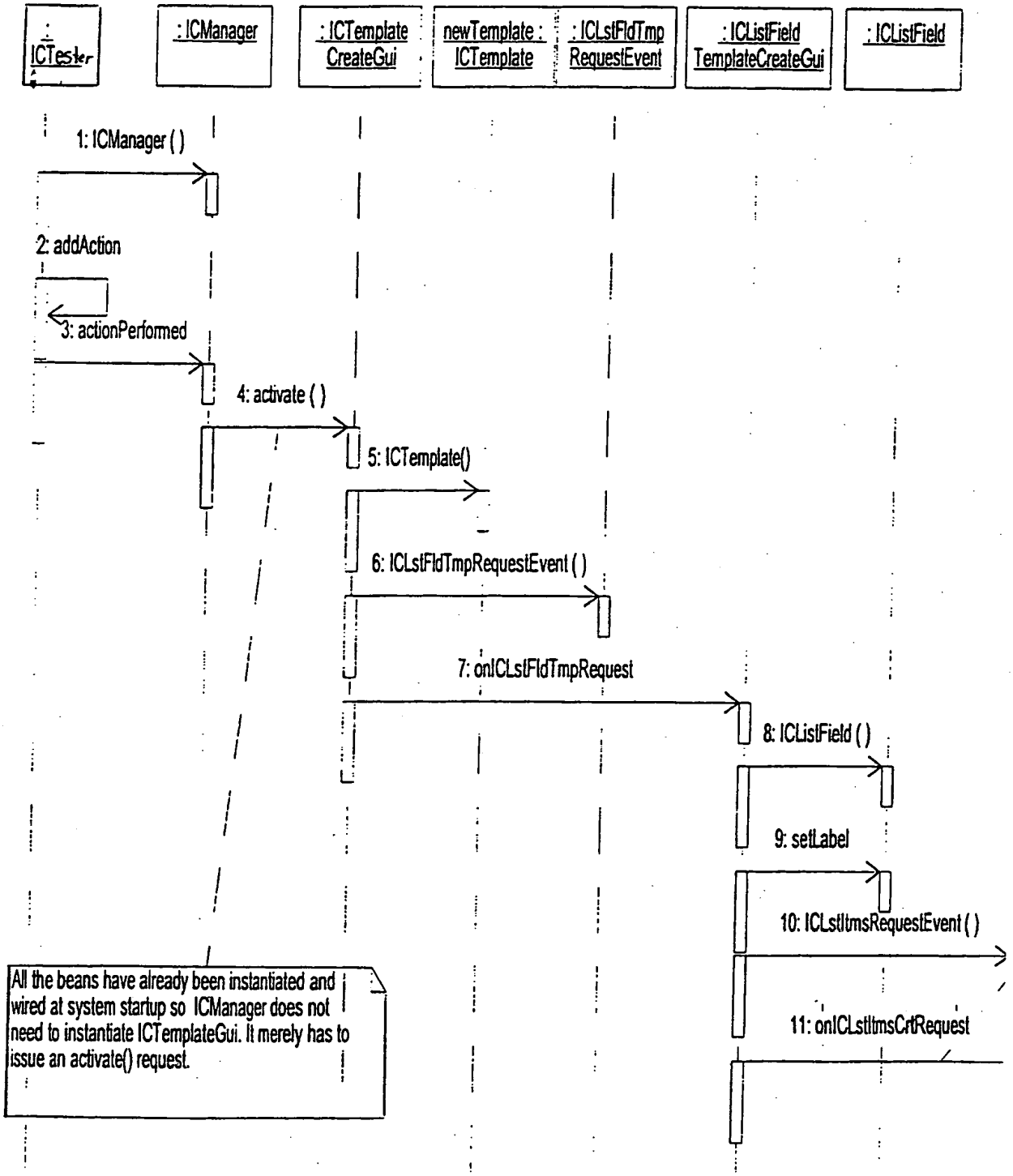

ICTester
: ICManager
: ICTemplate CreateGui
newTemplate : ICTemplate
: ICLstFldTmp RequestEvent
: ICListField TemplateCreateGui
: ICListField
1: ICManager ( )
2: addAction
3: actionPerformed
4: activate ( )
5: ICTemplate()
6: ICLstFldTmpRequestEvent ( )
7: onICLstFldTmpRequest
8: ICListField ( )
9: setLabel
10: ICLstItmsRequestEvent ( )
11: onICLstItmsCrtRequest
All the beans have already been instantiated and wired at system startup so ICManager does not need to instantiate ICTemplateGui. It merely has to issue an activate() request.

Fig. 14a

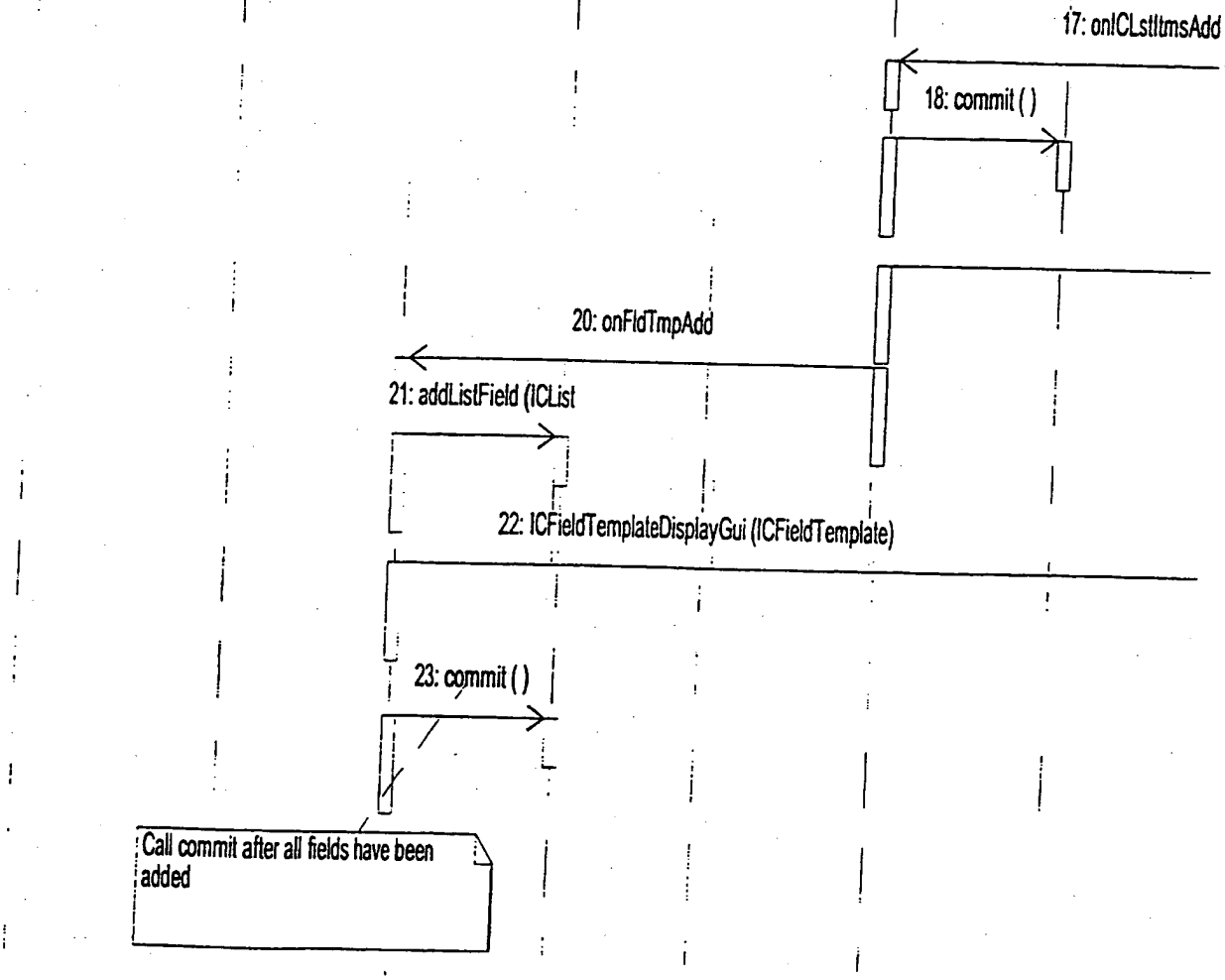

17: onICLstItmsAdd
18: commit ( )
20: onFldTmpAdd
21: addListField (ICList
22: ICFieldTemplateDisplayGui (ICFieldTemplate)
23: commit ( )
Call commit after all fields have been added

Fig. 14b

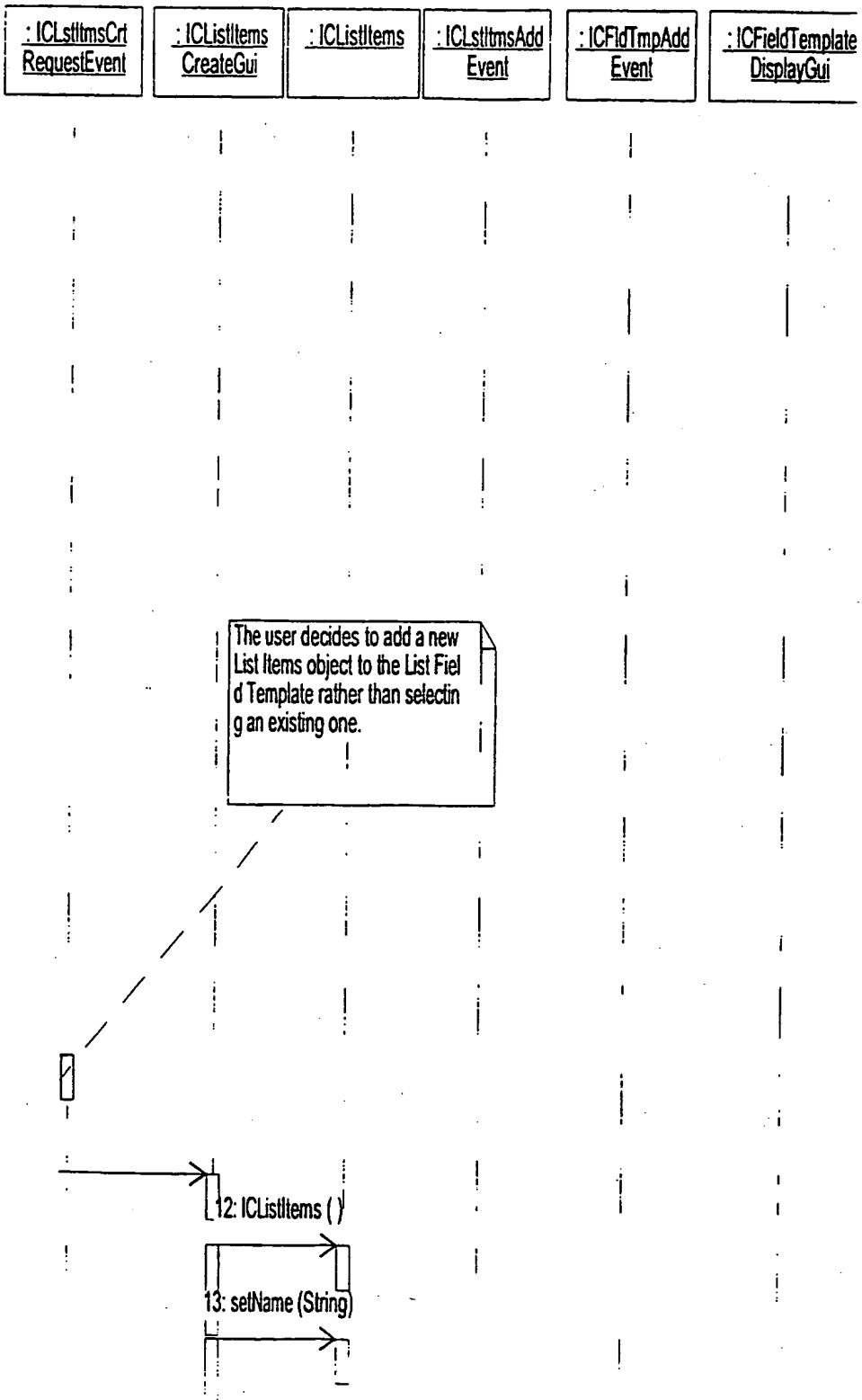

: ICLstItmsCrt RequestEvent
: ICListItems CreateGui
: ICListItems
: ICLstItmsAdd Event
: ICFldTmpAdd Event
: ICFieldTemplate DisplayGui
The user decides to add a new List Items object to the List Fiel d Template rather than selectin g an existing one.
12: ICListItems ( )
13: setName (String)

Fig. 14c

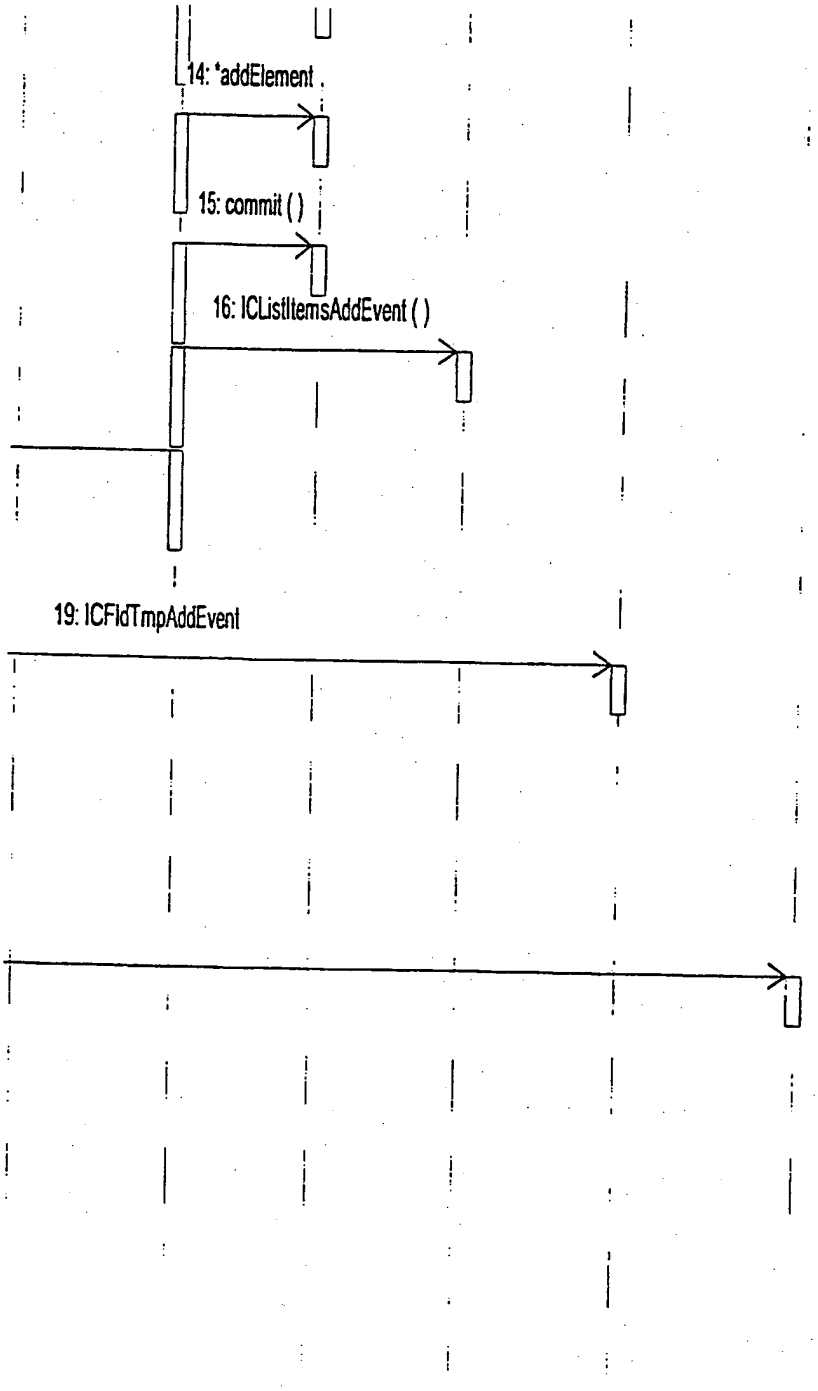

14: *addElement .
15: commit ( )
16: ICListItemsAddEvent ( )
19: ICFldTmpAddEvent

Fig. 14d

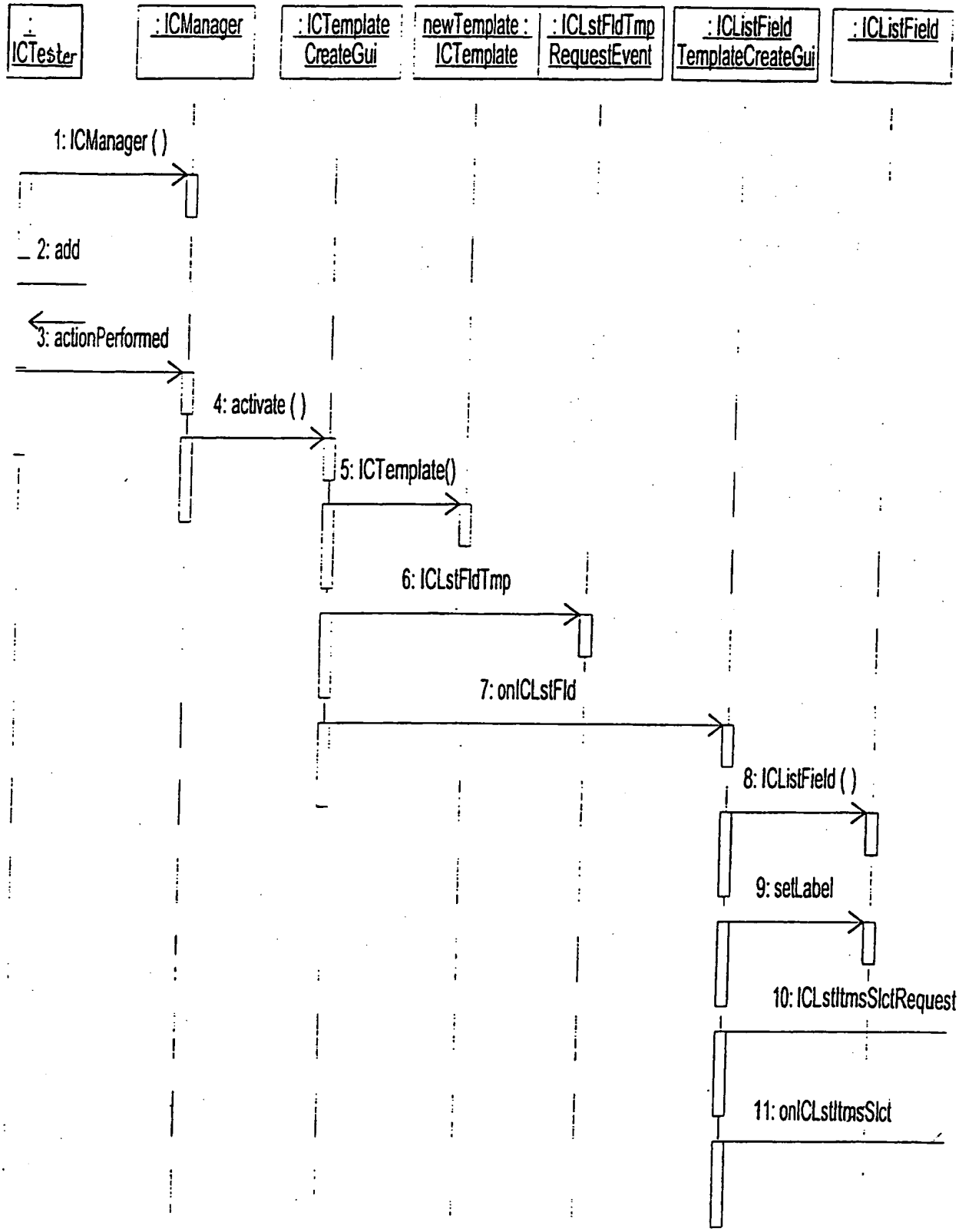

: ICTester
: ICManager
: ICTemplate CreateGui
newTemplate : ICTemplate
: ICLstFldTmp RequestEvent
: ICListField TemplateCreateGui
: ICListField
1: ICManager ( )
2: add
3: actionPerformed
4: activate ( )
5: ICTemplate()
6: ICLstFldTmp
7: onICLstFld
8: ICListField ( )
9: setLabel
10: ICLstItmsSlctRequest
11: onICLstItmsSlct

Fig. 15a

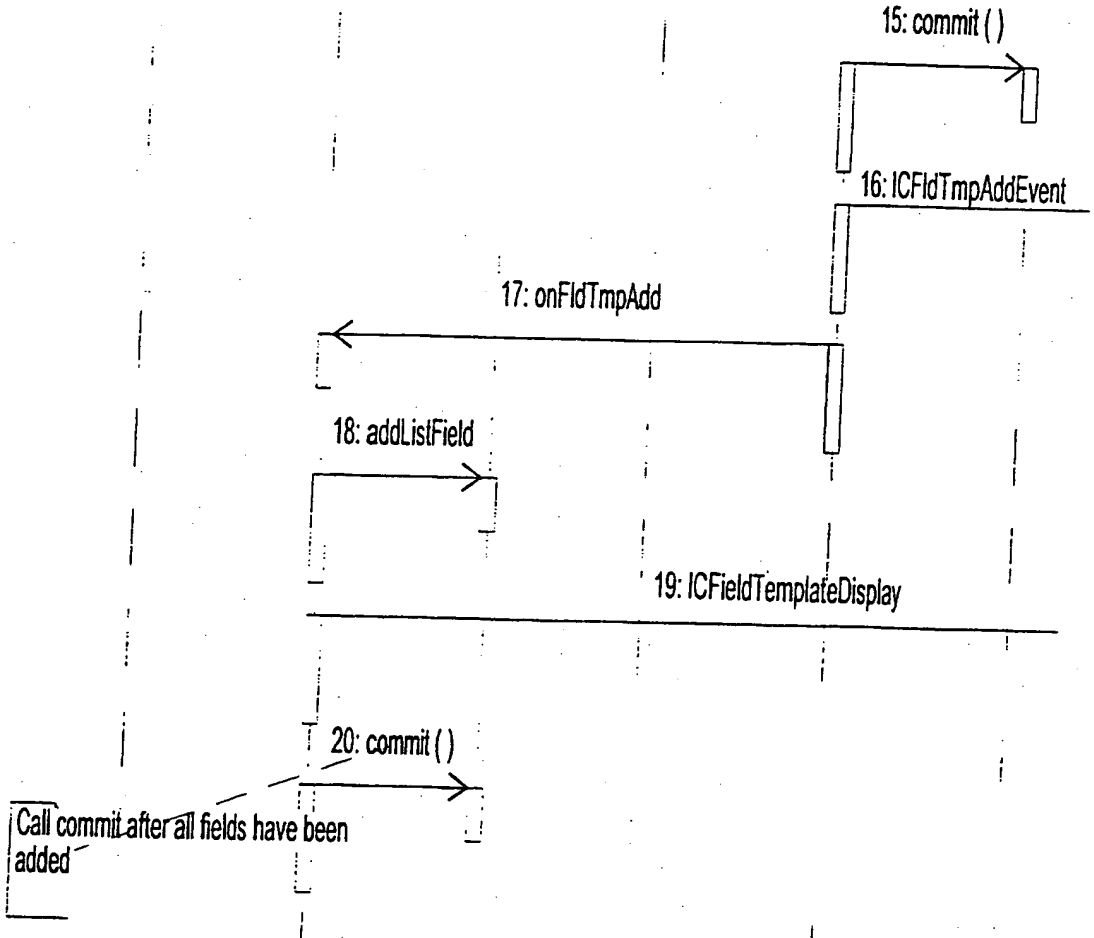

15: commit ( )
16: ICFldTmpAddEvent
17: onFldTmpAdd
18: addListField
19: ICFieldTemplateDisplay
20: commit ( )
Call commit after all fields have been added

Fig. 15b

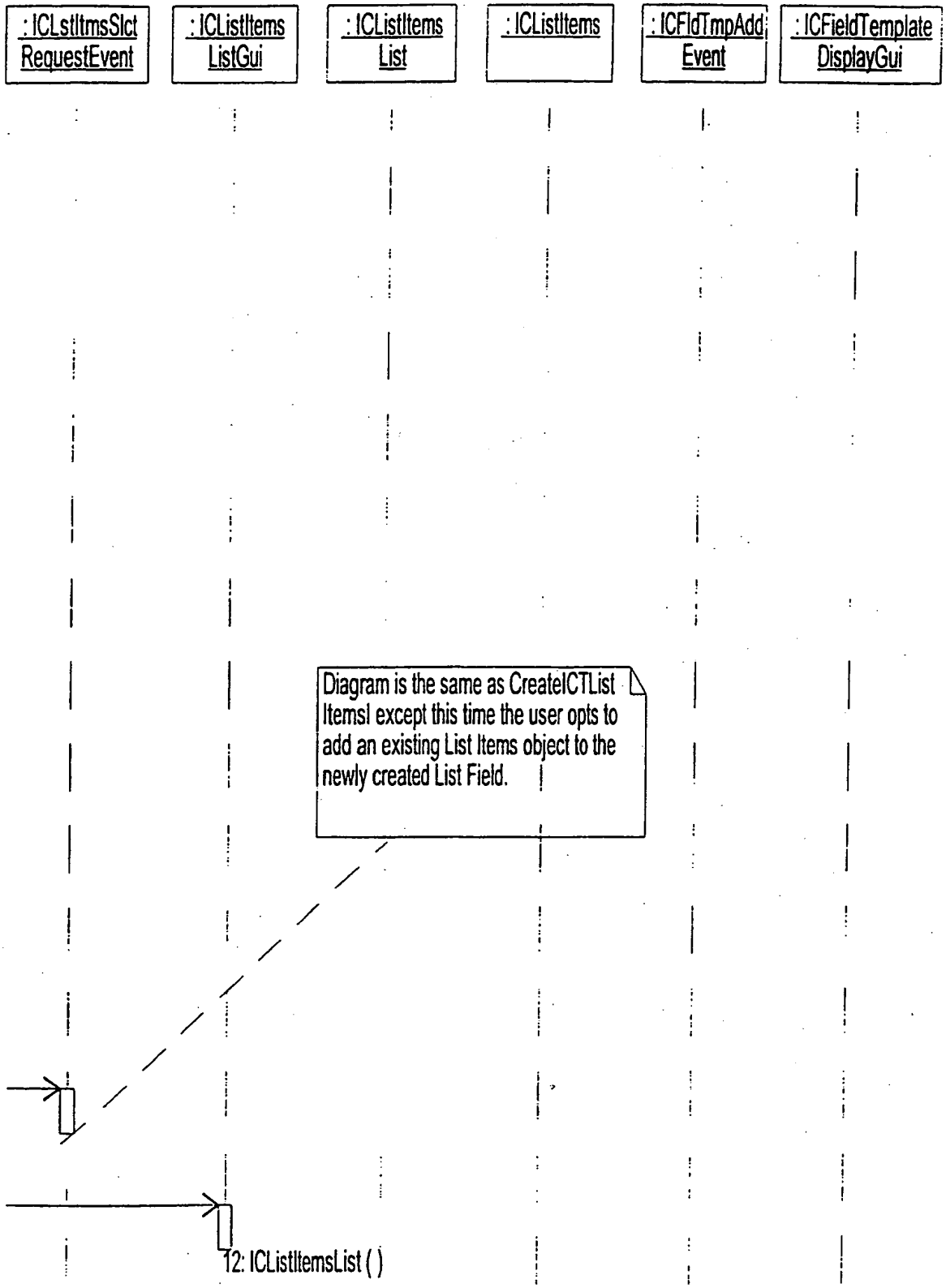

: ICLstItmsSlct RequestEvent
: IClistItems ListGui
: IClistItems List
: IClistItems
: ICFldTmpAdd Event
: ICFieldTemplate DisplayGui
Diagram is the same as CreateICTList Items! except this time the user opts to add an existing List Items object to the newly created List Field.
12: ICListItemsList ( )

Fig. 15c

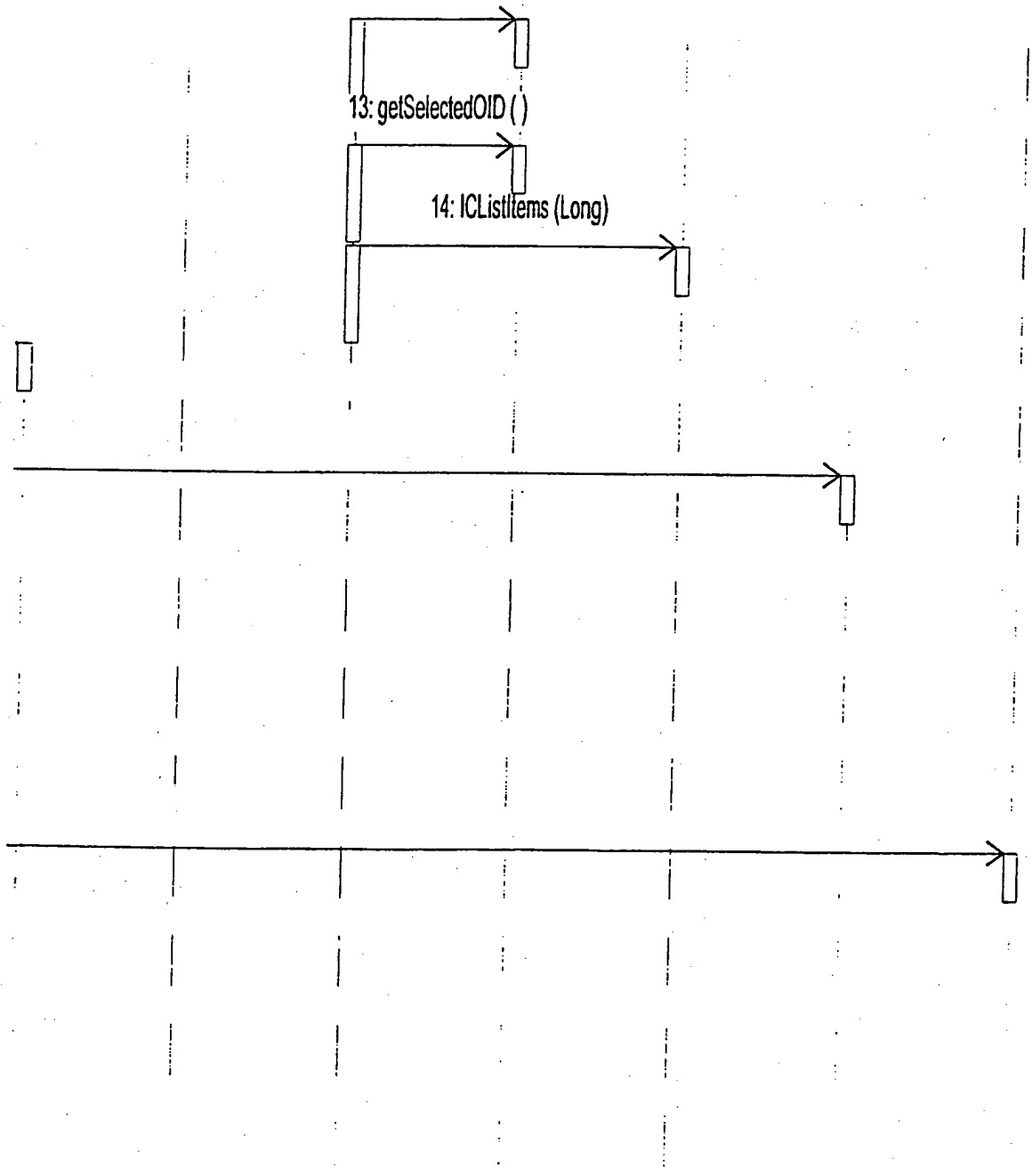


Fig. 15d

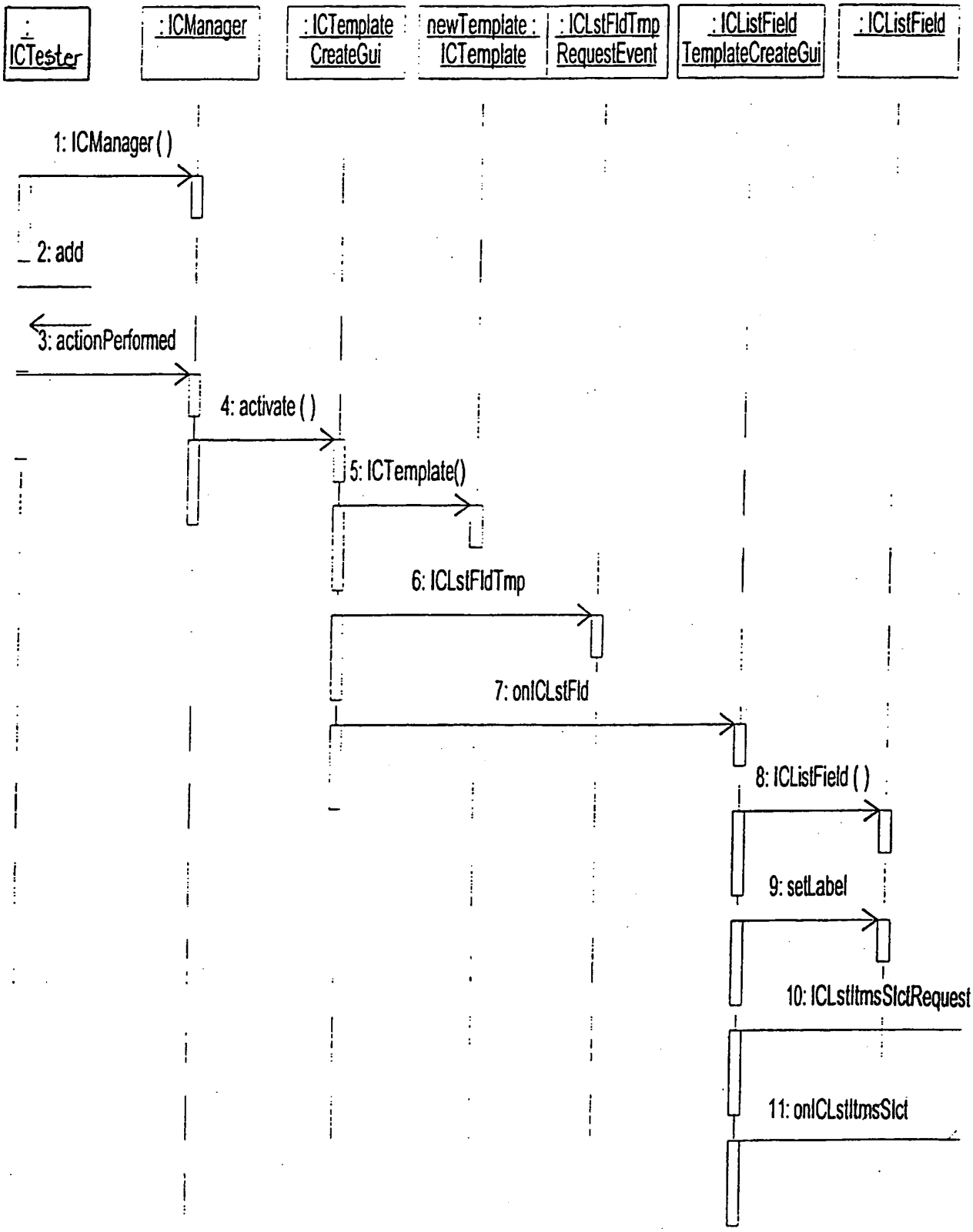

: ICTester
: ICManager
: ICTemplate CreateGui
newTemplate : ICTemplate
: ICLstFldTmp RequestEvent
: ICListField TemplateCreateGui
: ICListField
1: ICManager ( )
2: add
3: actionPerformed
4: activate ( )
5: ICTemplate()
6: ICLstFldTmp
7: onICLstFld
8: ICListField ( )
9: setLabel
10: ICLstItmsSlctRequest
11: onICLstItmsSlct

Fig. 16a

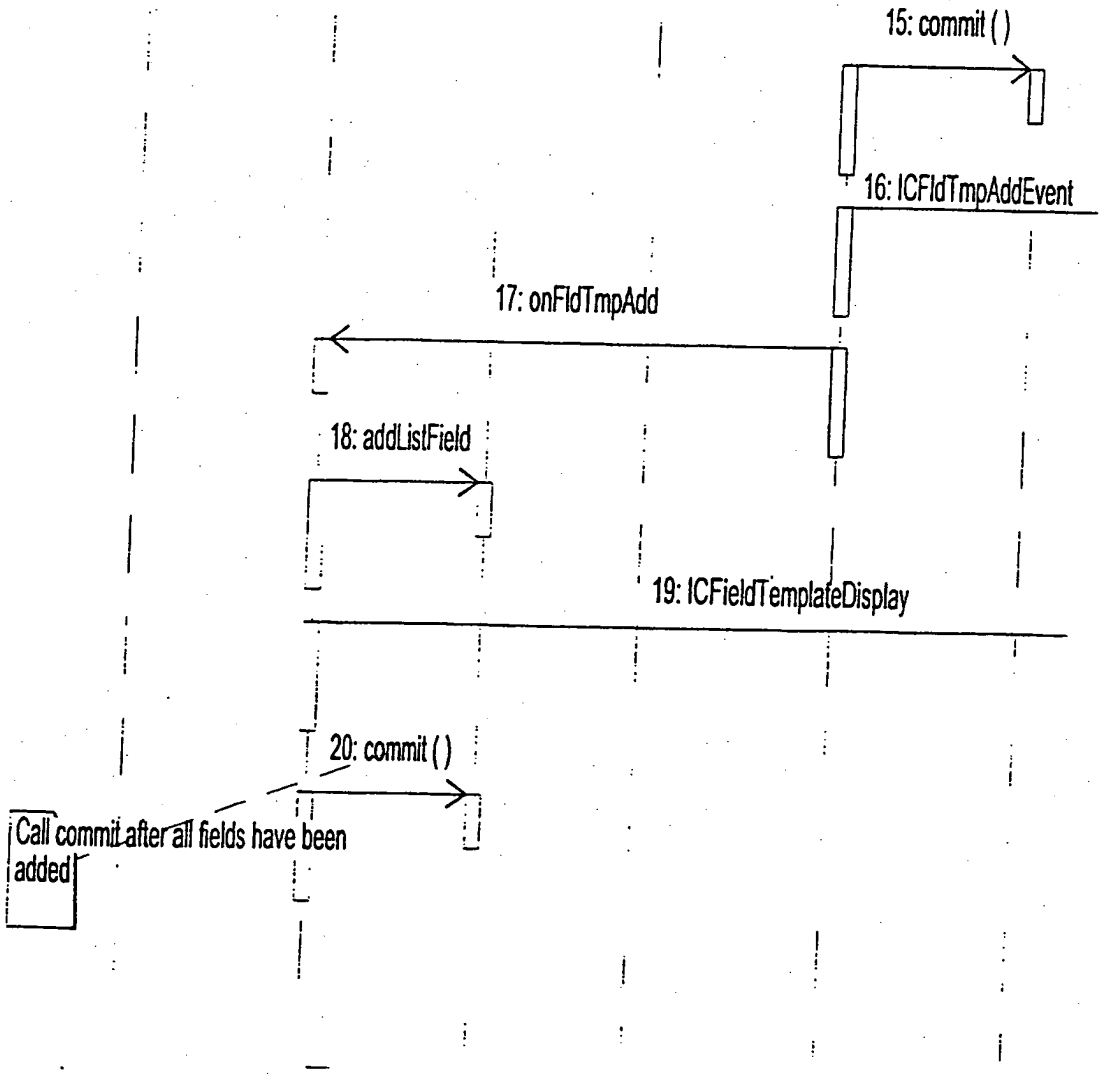

15: commit ( )
16: ICFldTmpAddEvent
17: onFldTmpAdd
18: addListField
19: ICFieldTemplateDisplay
20: commit ( )
Call commit after all fields have been added

Fig. 16b

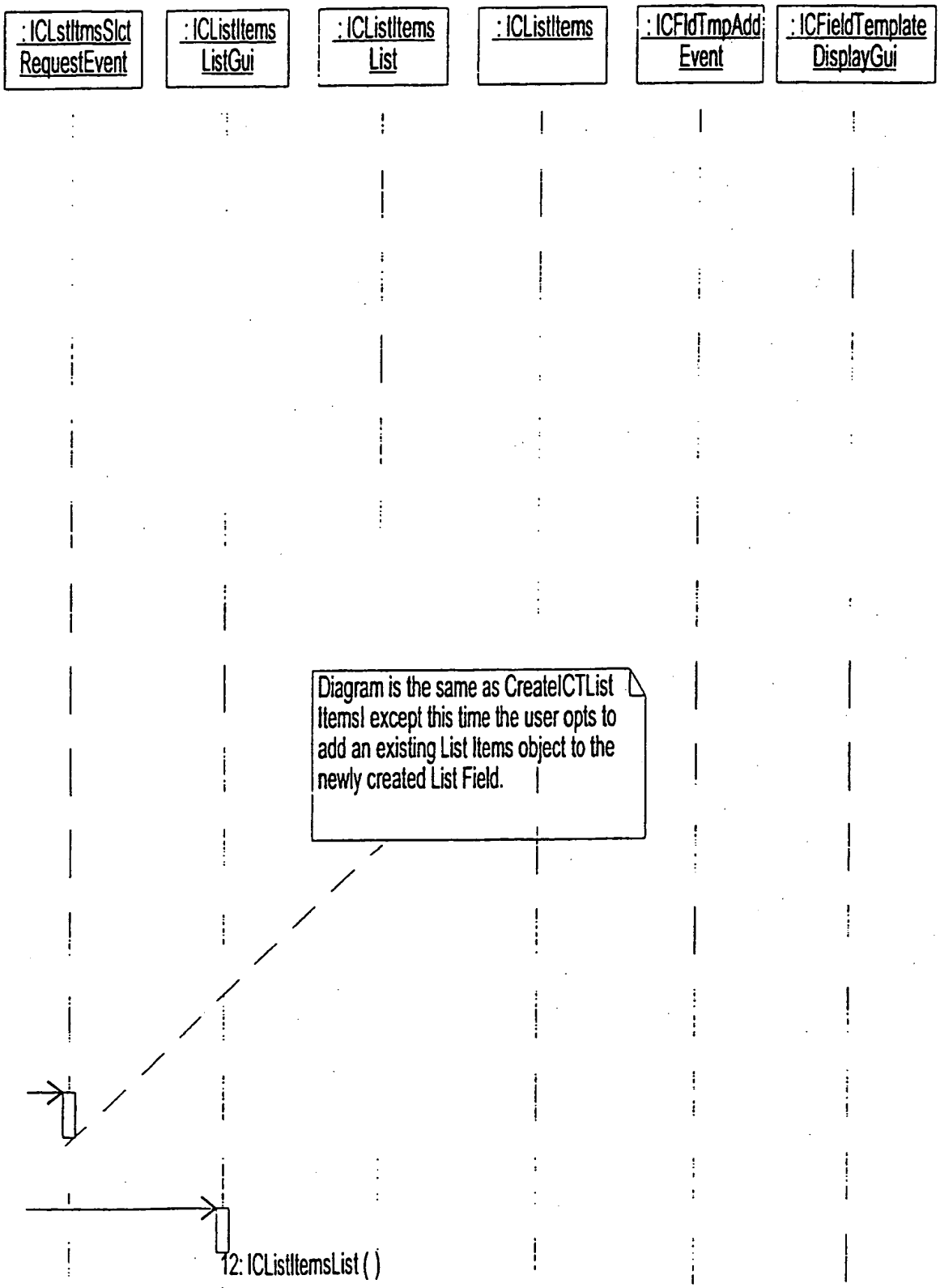

: ICLstItmsSlct RequestEvent
: IClistItems ListGui
: IClistItems List
: IClistItems
: ICFldTmpAdd Event
: ICFieldTemplate DisplayGui
Diagram is the same as CreateICTList Items! except this time the user opts to add an existing List Items object to the newly created List Field.
12: IClistItemsList ( )

Fig. 16c

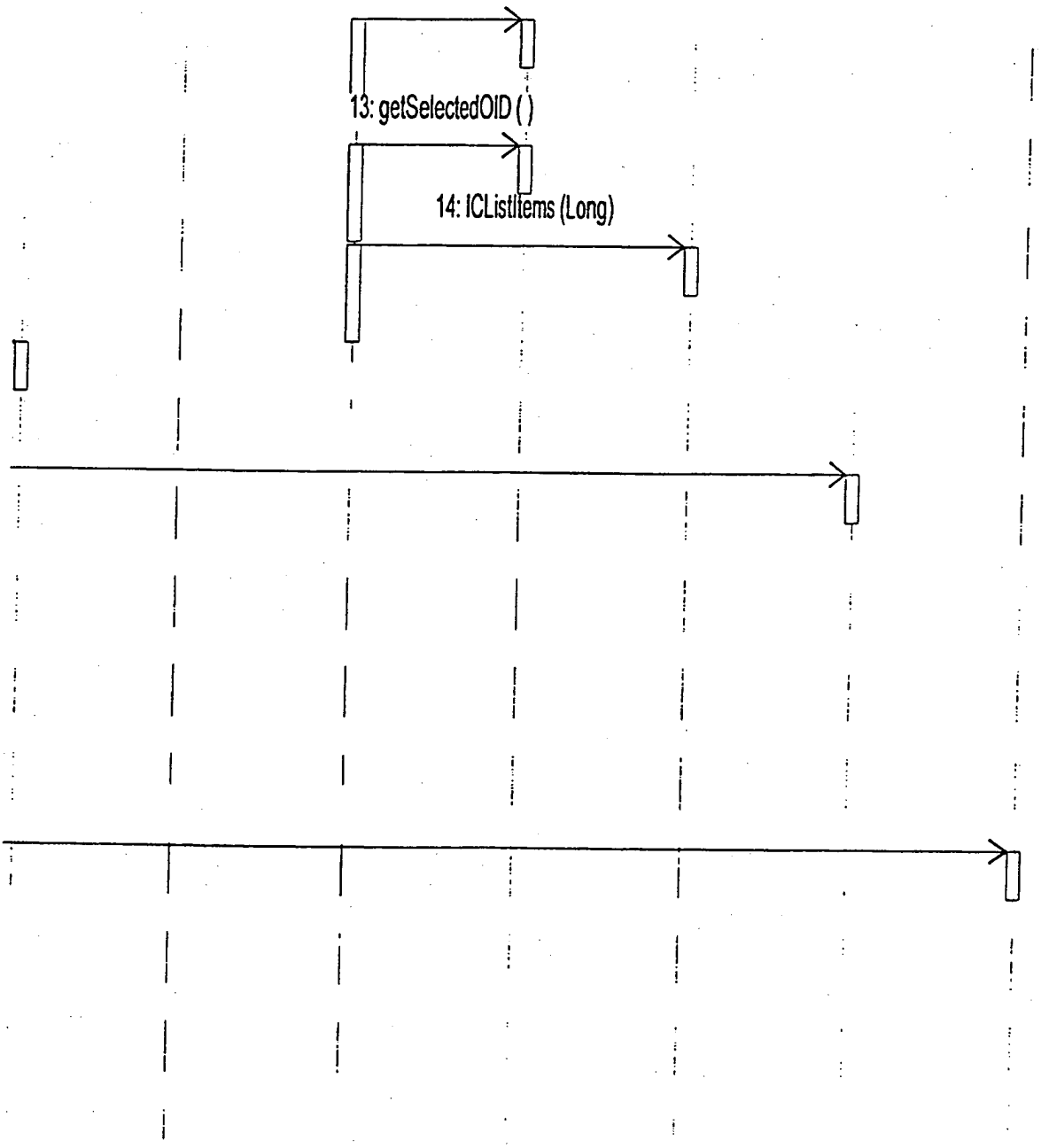


Fig. 16d

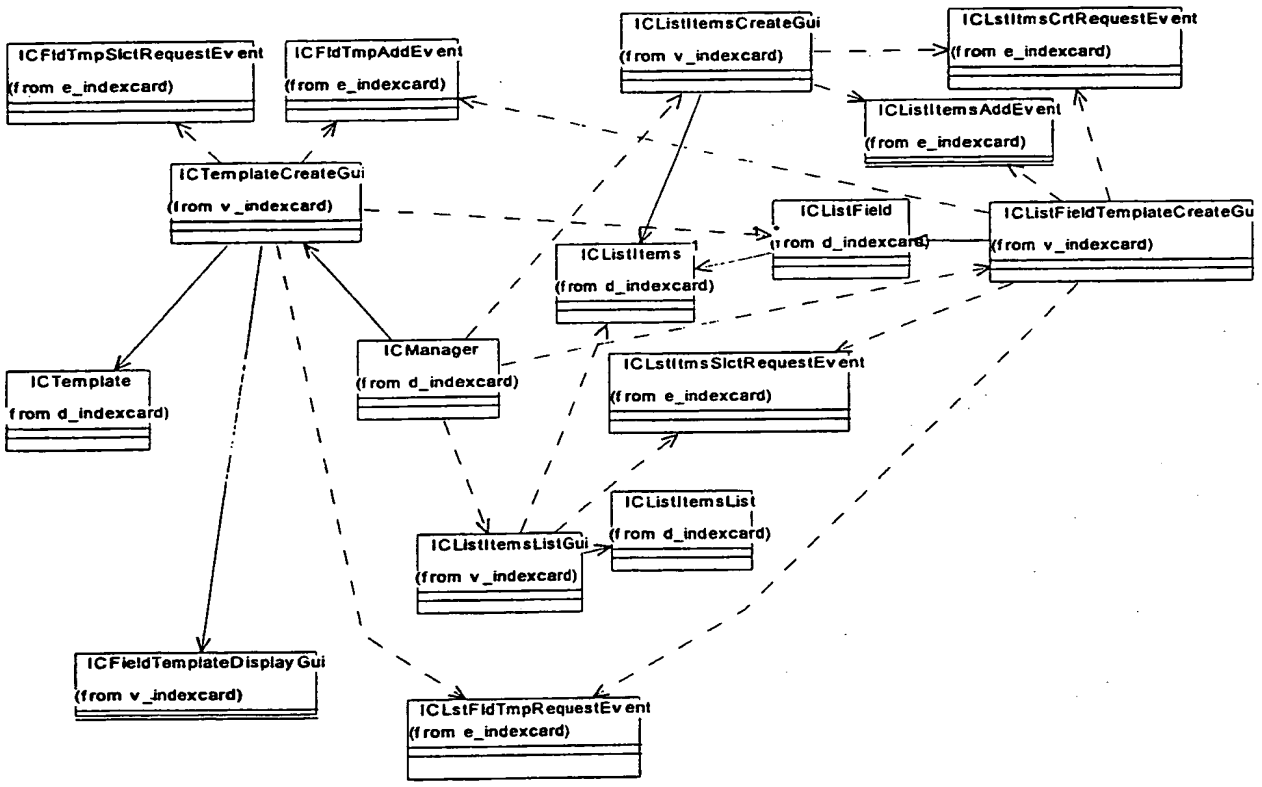

ICFldTmpSlctRequestEvent
(from e_indexcard)
ICFldTmpAddEvent
(from e_indexcard)
ICListItemsCreateGui
(from v_indexcard)
ICLstItmsCrtRequestEvent
(from e_indexcard)
ICListItemsAddEvent
(from e_indexcard)
ICTemplateCreateGui
(from v_indexcard)
ICListField
(from d_indexcard)
ICListFieldTemplateCreateGu
(from v_indexcard)
ICListItems
(from d_indexcard)
ICManager
(from d_indexcard)
ICTemplate
from d_indexcard)
ICLstItmsSlctRequestEvent
(from e_indexcard)
ICListItemsList
(from d_indexcard)
ICListItemsListGui
(from v_indexcard)
ICFieldTemplateDisplayGui
(from v_indexcard)
ICLstFldTmpRequestEvent
(from e_indexcard)

Fig. 17

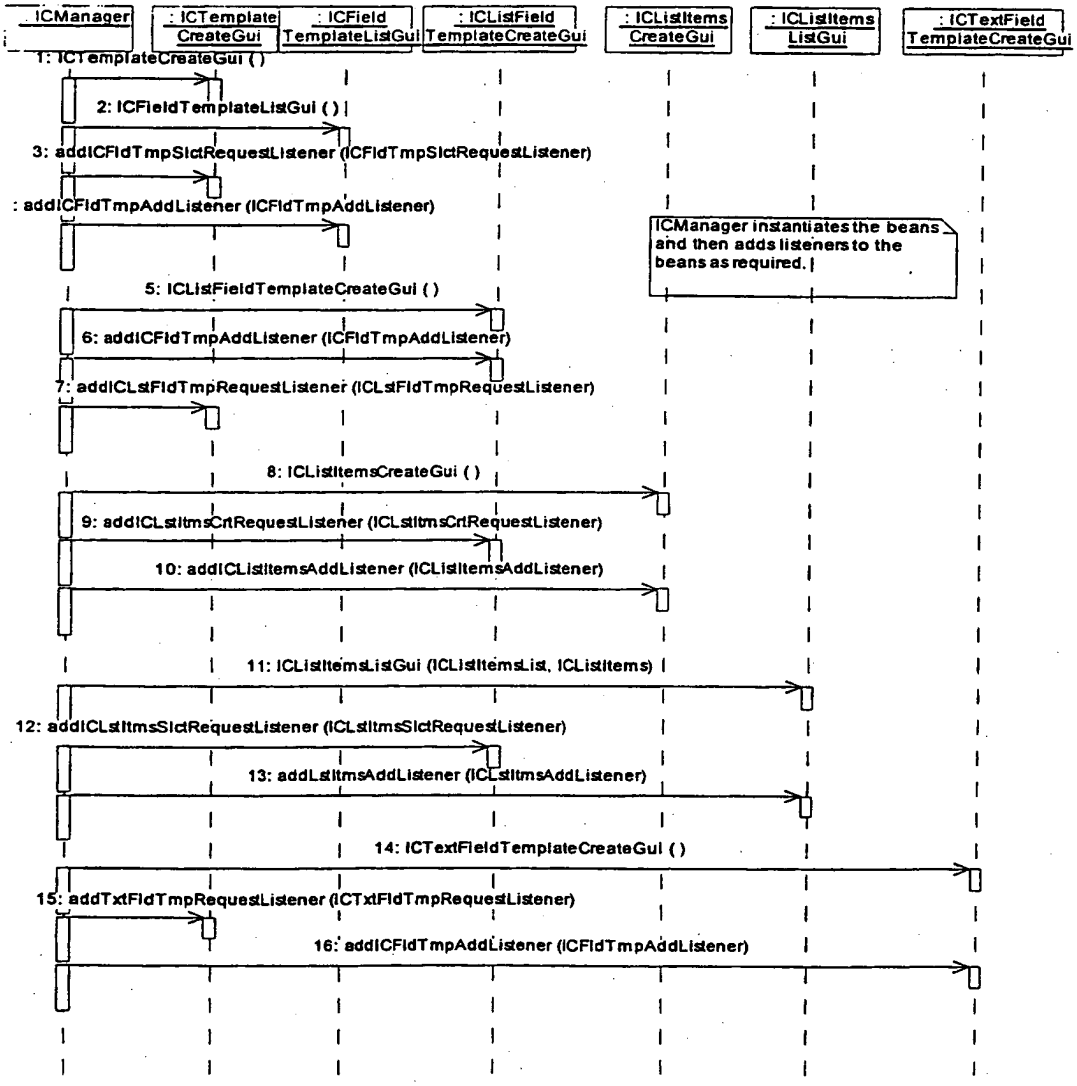

ICManager
: ICTemplate CreateGui
: ICField TemplateListGui
: ICListField TemplateCreateGui
: ICListItems CreateGui
: ICListItems ListGui
: ICTextField TemplateCreateGui
1: ICTemplateCreateGui ( )
2: ICFieldTemplateListGui ( )
3: addICFldTmpSlctRequestListener (ICFldTmpSlctRequestListener)
: addICFldTmpAddListener (ICFldTmpAddListener)
ICManager instantiates the beans and then adds listeners to the beans as required.
5: ICListFieldTemplateCreateGui ( )
6: addICFldTmpAddListener (ICFldTmpAddListener)
7: addICLstFldTmpRequestListener (ICLstFldTmpRequestListener)
8: ICListItemsCreateGui ( )
9: addICLstItmsCrtRequestListener (ICLstItmsCrtRequestListener)
10: addICListItemsAddListener (ICListItemsAddListener)
11: ICListItemsListGui (ICListItemsList, ICListItems)
12: addICLstItmsSlctRequestListener (ICLstItmsSlctRequestListener)
13: addLstItmsAddListener (ICLstItmsAddListener)
14: ICTextFieldTemplateCreateGui ( )
15: addTxtFldTmpRequestListener (ICTxtFldTmpRequestListener)
16: addICFldTmpAddListener (ICFldTmpAddListener)

Fig. 18

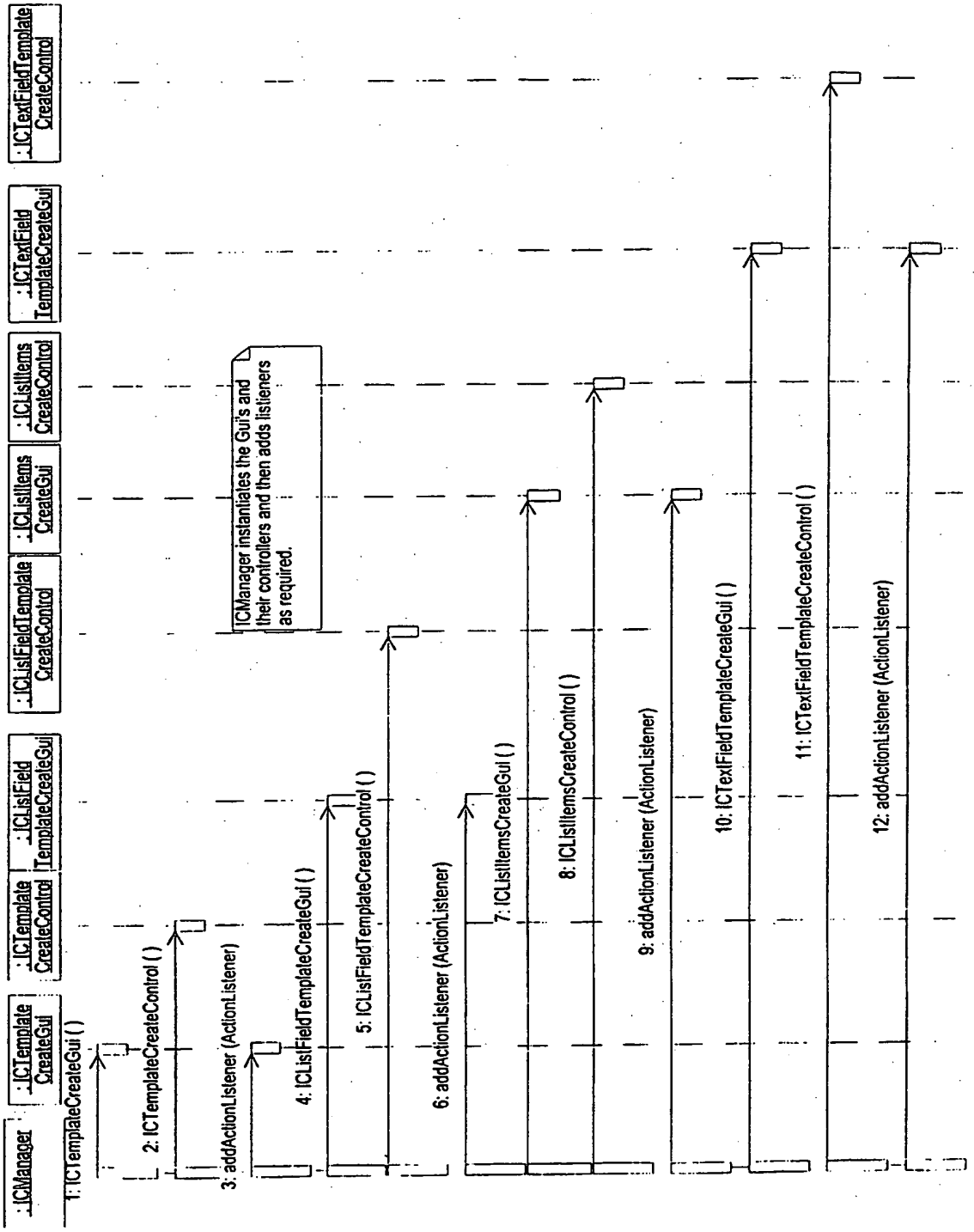

:ICManager
:ICTemplate CreateGui
:ICTemplate CreateControl
:ICListField TemplateCreateGui
:ICListFieldTemplate CreateControl
:ICListItems CreateGui
:ICListItems CreateControl
:ICTextField TemplateCreateGui
:ICTextFieldTemplate CreateControl
1: ICTemplateCreateGui ( )
2: ICTemplateCreateControl ( )
3: addActionListener (ActionListener)
4: ICListFieldTemplateCreateGui ( )
5: ICListFieldTemplateCreateControl ( )
6: addActionListener (ActionListener)
7: ICListItemsCreateGui ( )
8: ICListItemsCreateControl ( )
9: addActionListener (ActionListener)
10: ICTextFieldTemplateCreateGui ( )
11: ICTextFieldTemplateCreateControl ( )
12: addActionListener (ActionListener)
ICManager instantiates the Gui's and their controllers and then adds listieners as required.

Fig. 19

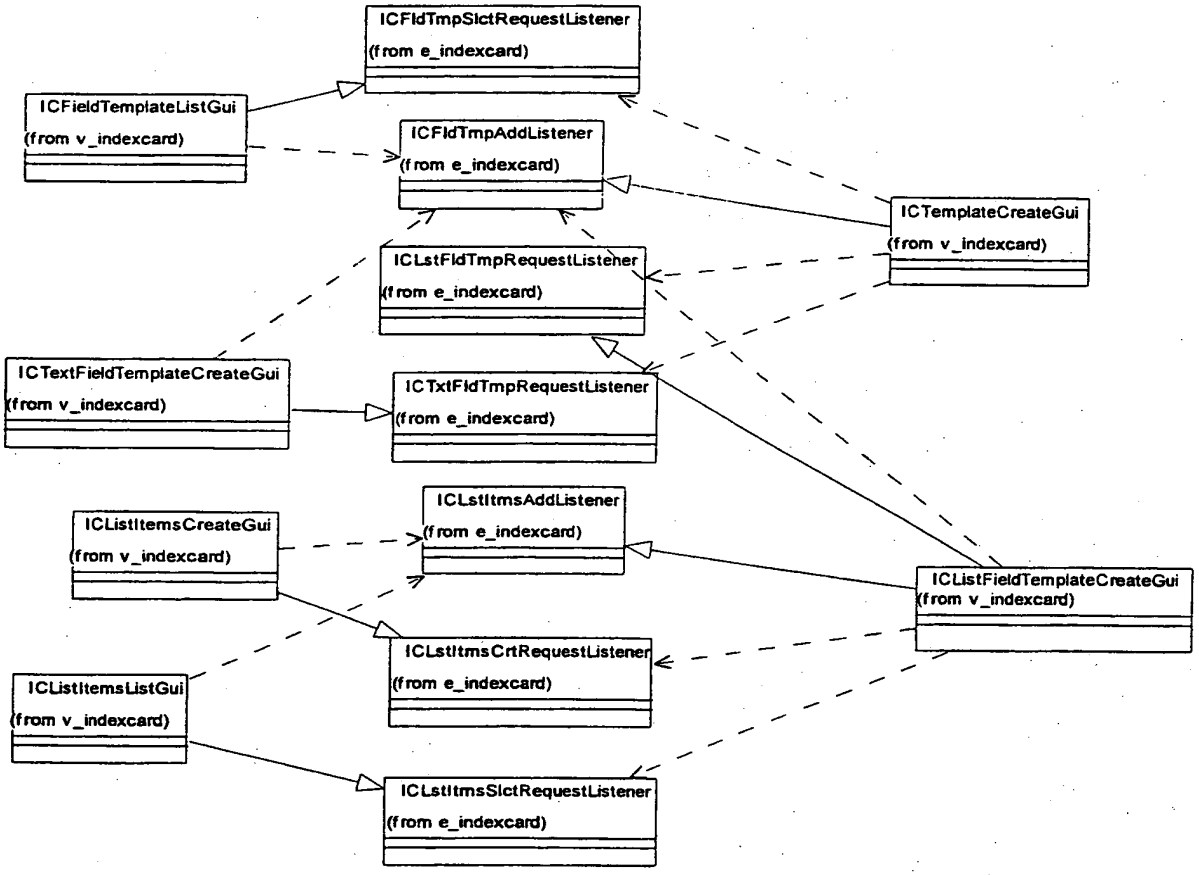

ICFldTmpSlctRequestListener
(from e_indexcard)
ICFieldTemplateListGui
(from v_indexcard)
ICFldTmpAddListener
(from e_indexcard)
ICTemplateCreateGui
(from v_indexcard)
ICLstFldTmpRequestListener
(from e_indexcard)
ICTextFieldTemplateCreateGui
(from v_indexcard)
ICTxtFldTmpRequestListener
(from e_indexcard)
ICLstItmsAddListener
(from e_indexcard)
ICListItemsCreateGui
(from v_indexcard)
ICListFieldTemplateCreateGui
(from v_indexcard)
ICLstItmsCrtRequestListener
(from e_indexcard)
ICListItemsListGui
(from v_indexcard)
ICLstItmsSlctRequestListener
(from e_indexcard)

Fig. 20

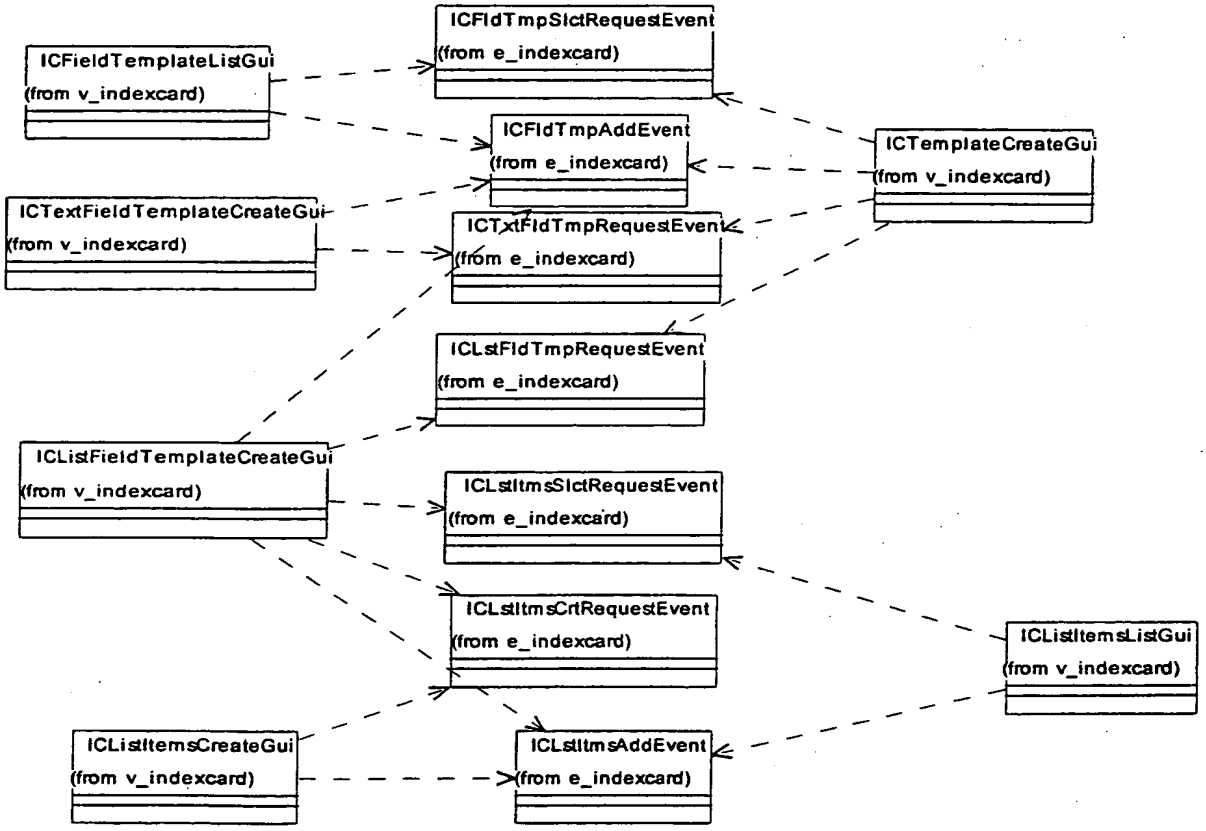

ICFieldTemplateListGui
(from v_indexcard)
ICFldTmpSlctRequestEvent
(from e_indexcard)
ICFldTmpAddEvent
(from e_indexcard)
ICTemplateCreateGui
(from v_indexcard)
ICTextFieldTemplateCreateGui
(from v_indexcard)
ICTxtFldTmpRequestEvent
(from e_indexcard)
ICLstFldTmpRequestEvent
(from e_indexcard)
ICListFieldTemplateCreateGui
(from v_indexcard)
ICLstItmsSlctRequestEvent
(from e_indexcard)
ICLstItmsCrtRequestEvent
(from e_indexcard)
ICListItemsListGui
(from v_indexcard)
ICListItemsCreateGui
(from v_indexcard)
ICLstItmsAddEvent
(from e_indexcard)

Fig. 21

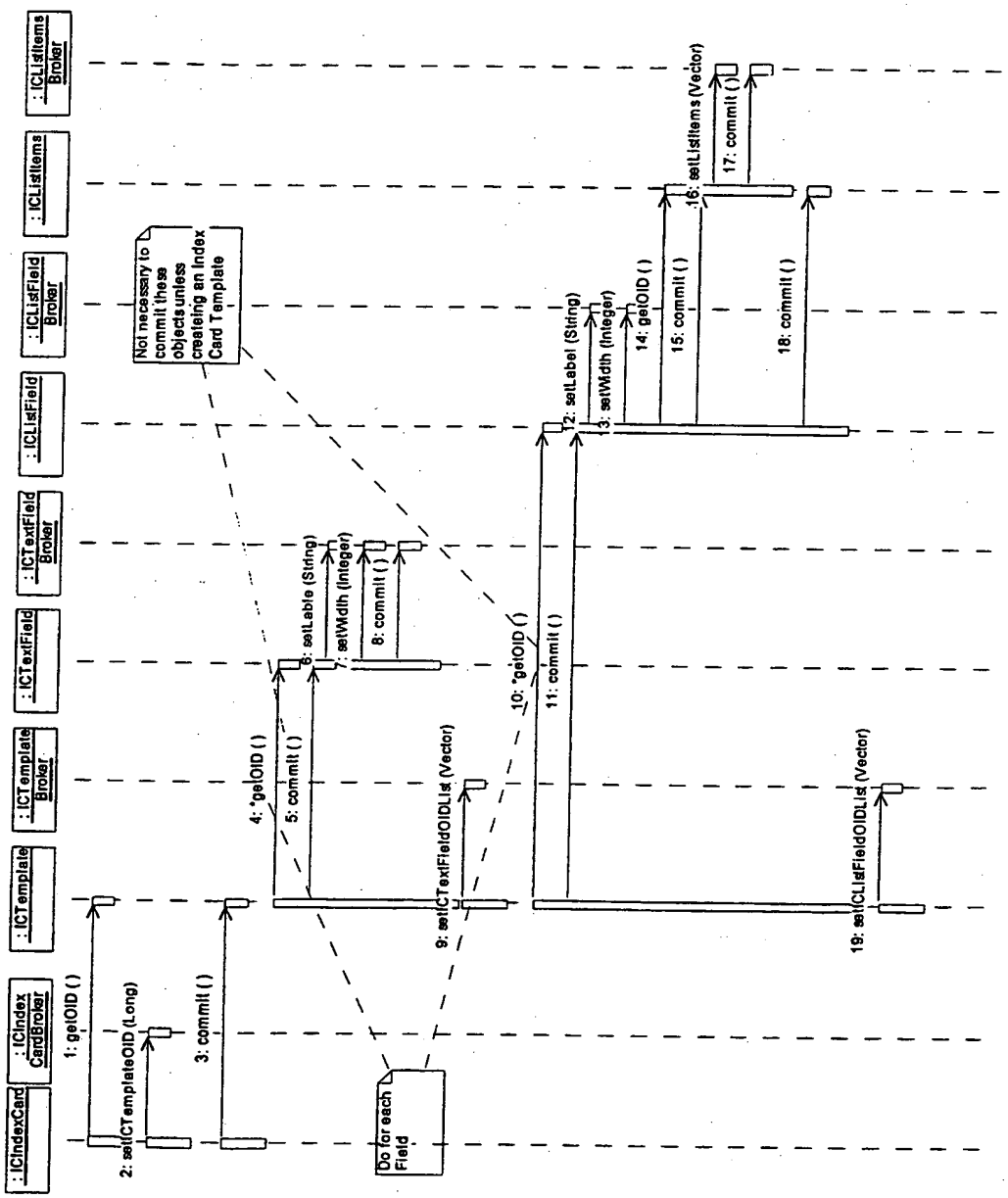

: ICIndexCard
: ICIndex CardBroker
: ICTemplate
: ICTemplate Broker
: ICTextField
: ICTextField Broker
: ICListField
: ICListField Broker
: ICListItems
: ICListItems Broker
1: getOID ( )
2: setICTemplateOID (Long)
3: commit ( )
4: *getOID ( )
5: commit ( )
6: setLable (String)
7: setWidth (Integer)
8: commit ( )
9: setICTextFieldOIDList (Vector)
10: *getOID ( )
11: commit ( )
12: setLabel (String)
13: setWidth (Integer)
14: getOID ( )
15: commit ( )
16: setListItems (Vector)
17: commit ( )
18: commit ( )
19: setICListFieldOIDList (Vector)
Not necessary to commit these objects unless creating an Index Card Template
Do for each Field

Fig. 22

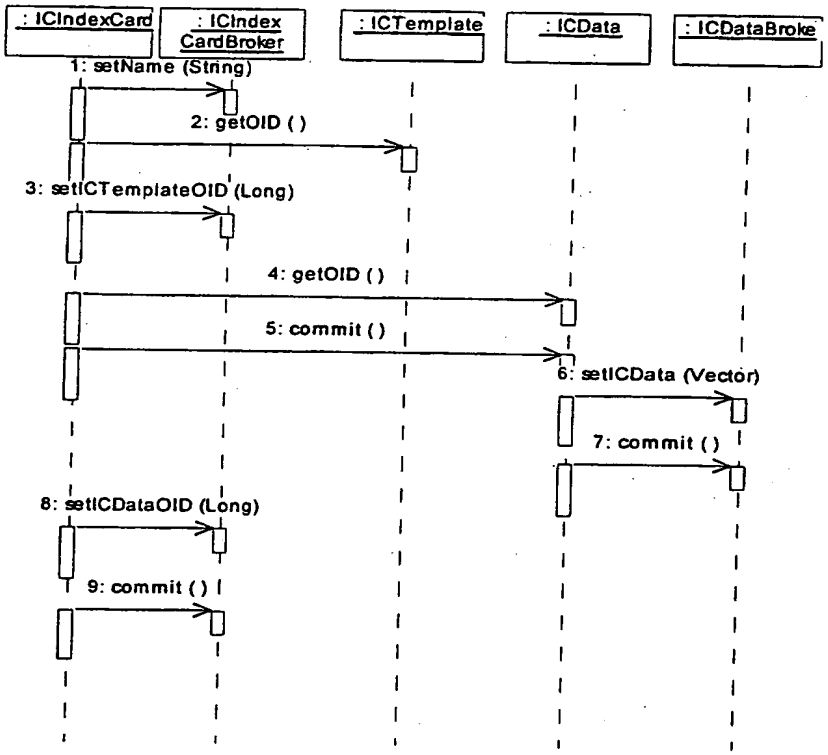

: ICIndexCard
: ICIndex CardBroker
: ICTemplate
: ICData
: ICDataBroke
1: setName (String)
2: getOID ( )
3: setICTemplateOID (Long)
4: getOID ( )
5: commit ( )
6: setICData (Vector)
7: commit ( )
8: setICDataOID (Long)
9: commit ( )

Fig. 23

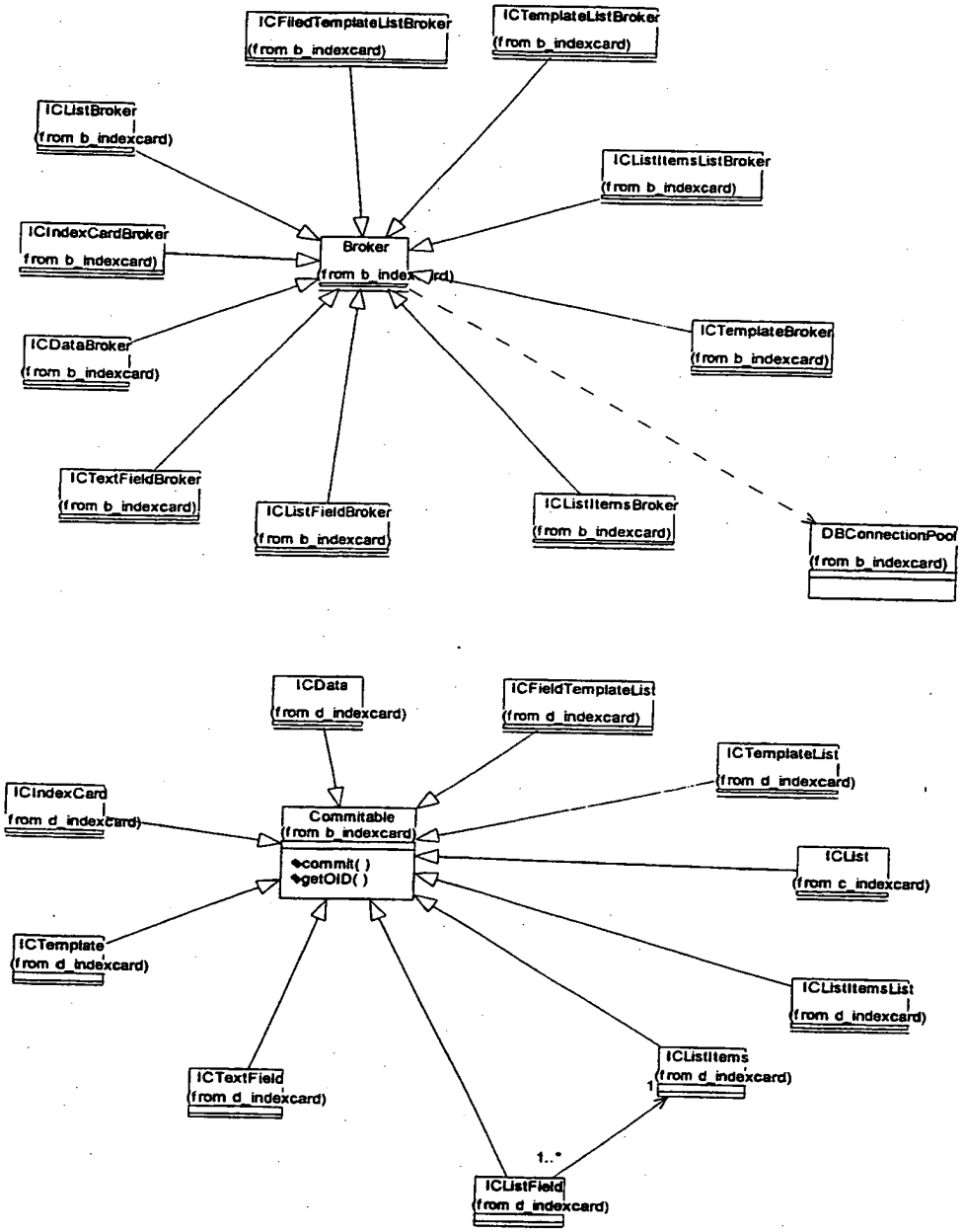

ICFiledTemplateListBroker
(from b_indexcard)
ICTemplateListBroker
(from b_indexcard)
ICListBroker
(from b_indexcard)
ICListItemsListBroker
(from b_indexcard)
ICIndexCardBroker
(from b_indexcard)
Broker
(from b_indexcard)
ICTemplateBroker
(from b_indexcard)
ICDataBroker
(from b_indexcard)
ICTextFieldBroker
(from b_indexcard)
ICListFieldBroker
(from b_indexcard)
ICListItemsBroker
(from b_indexcard)
DBConnectionPool
(from b_indexcard)
ICData
(from d_indexcard)
ICFieldTemplateList
(from d_indexcard)
ICTemplateList
(from d_indexcard)
ICIndexCard
(from d_indexcard)
Commitable
(from b_indexcard)
commit( )
getOID( )
ICList
(from c_indexcard)
ICTemplate
(from d_indexcard)
ICListItemsList
(from d_indexcard)
ICListItems
(from d_indexcard)
ICTextField
(from d_indexcard)
1
1..*
ICListField
(from d_indexcard)

Fig. 24

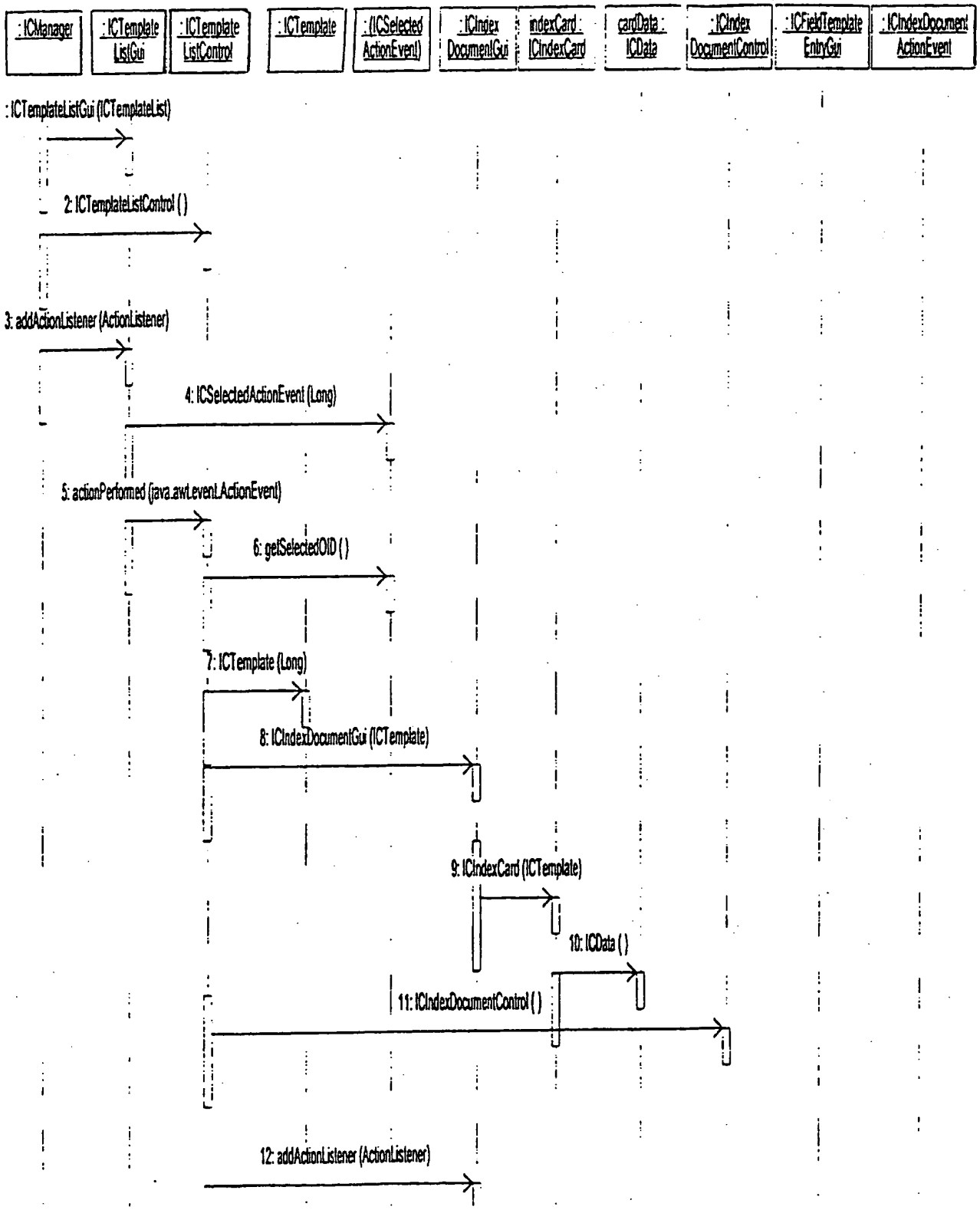

: ICManager
: ICTemplate ListGui
: ICTemplate ListControl
: ICTemplate
: ICSelected ActionEvent()
: ICIndex DocumentGui
indexCard : ICIndexCard
cardData : ICData
: ICIndex DocumentControl
: ICFieldTemplate EntryGui
: ICIndexDocument ActionEvent
: ICTemplateListGui (ICTemplateList)
2: ICTemplateListControl ( )
3: addActionListener (ActionListener)
4: ICSelectedActionEvent (Long)
5: actionPerformed (java.awt.event.ActionEvent)
6: getSelectedOID ( )
7: ICTemplate (Long)
8: ICIndexDocumentGui (ICTemplate)
9: ICIndexCard (ICTemplate)
10: ICData ( )
11: ICIndexDocumentControl ( )
12: addActionListener (ActionListener)

Fig. 25a

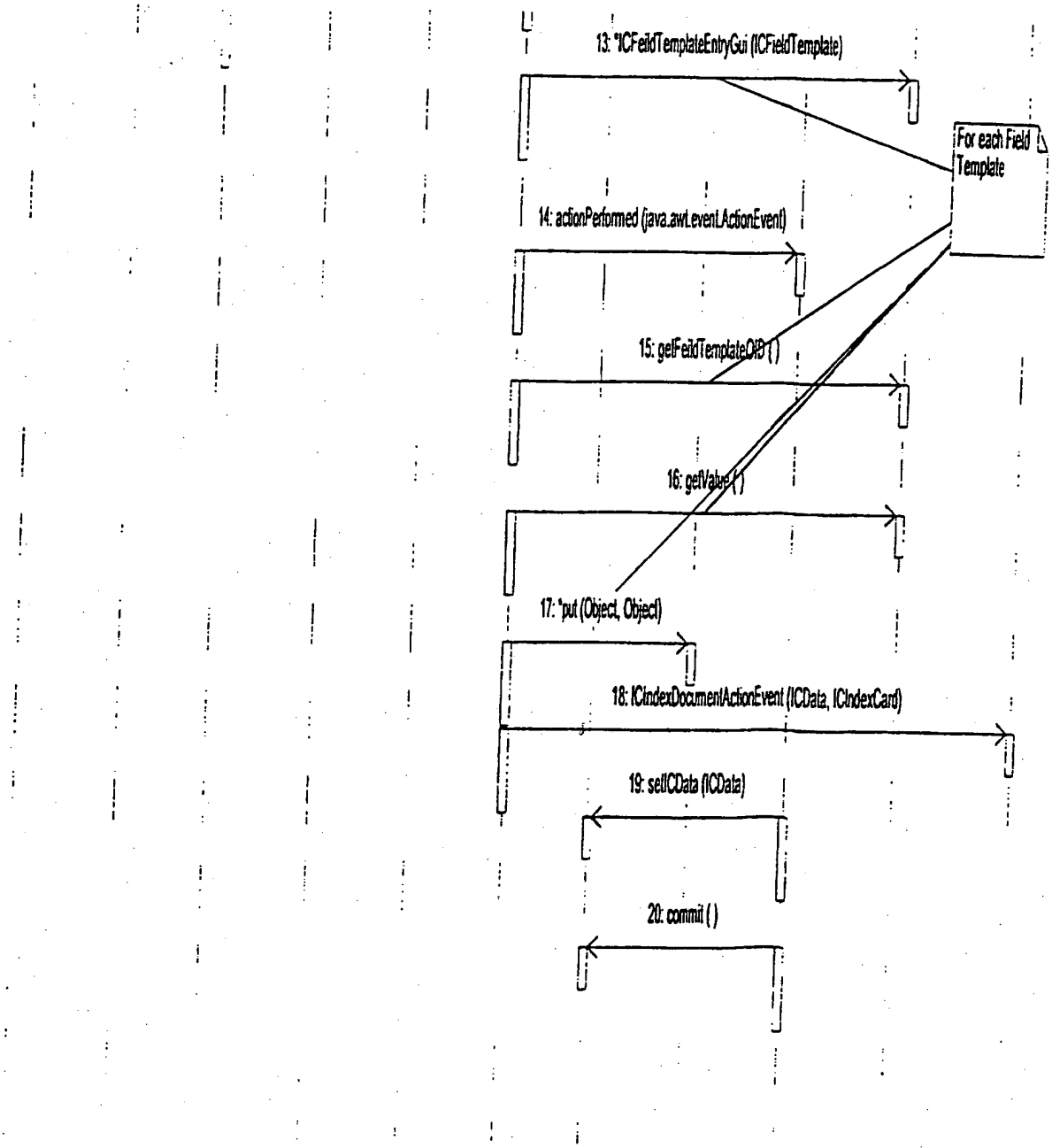

13: *ICFeildTemplateEntryGui (ICFieldTemplate)
For each Field Template
14: actionPerformed (java.awt.event.ActionEvent)
15: getFeildTemplateOID ( )
16: getValue ( )
17: *put (Object, Object)
18: ICIndexDocumentActionEvent (ICData, ICIndexCard)
19: setICData (ICData)
20: commit ( )

Fig. 25b

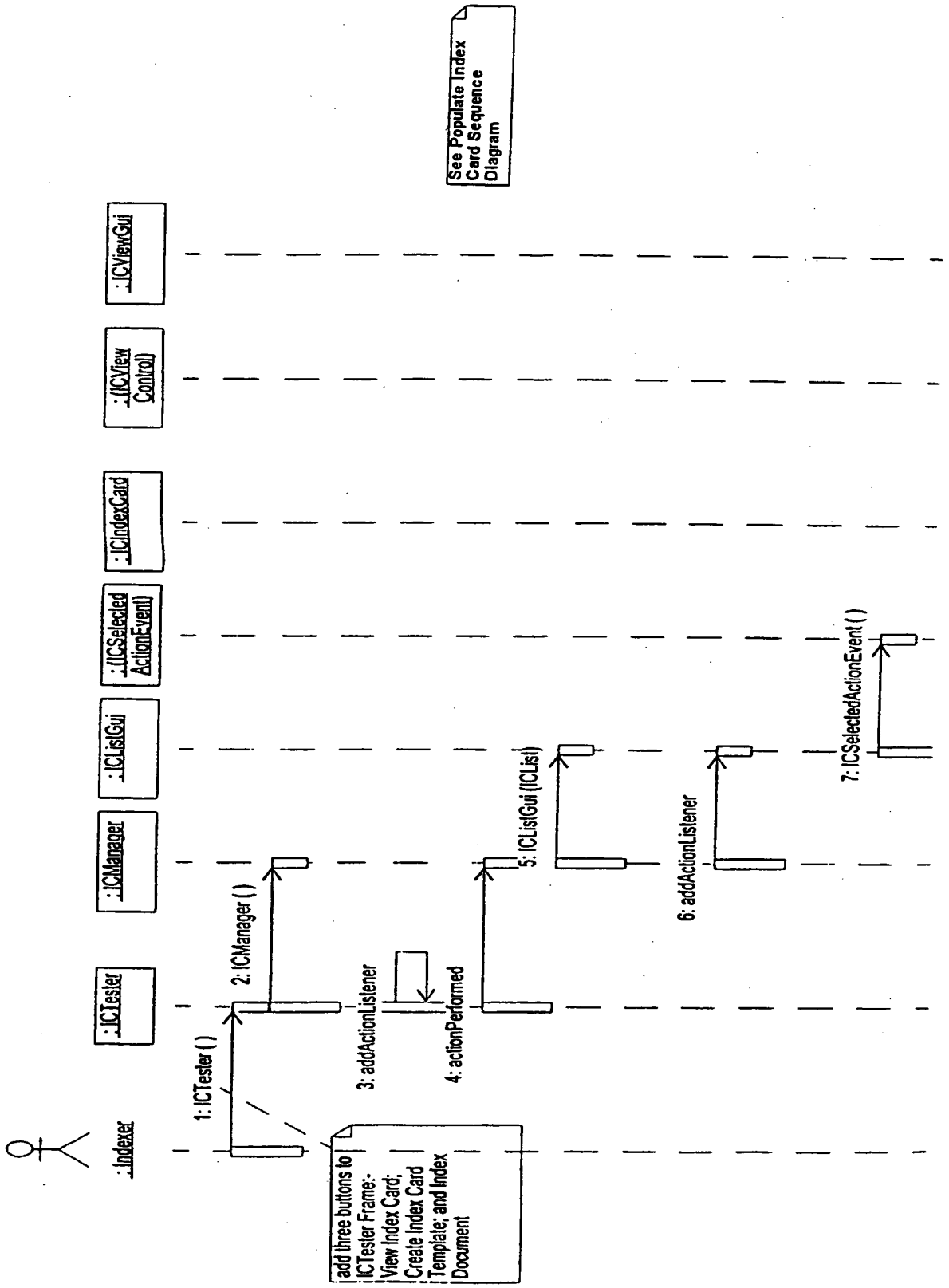

:Indexer
:ICTester
:ICManager
:ICListGui
:ICSelectedActionEvent
:ICIndexCard
:ICView Control
:ICViewGui
See Populate Index Card Sequence Diagram
1: ICTester ( )
2: ICManager ( )
3: addActionListener
4: actionPerformed
5: ICListGui (ICList)
6: addActionListener
7: ICSelectedActionEvent ( )
add three buttons to ICTester Frame:- View Index Card; Create Index Card Template; and Index Document

Fig. 26a

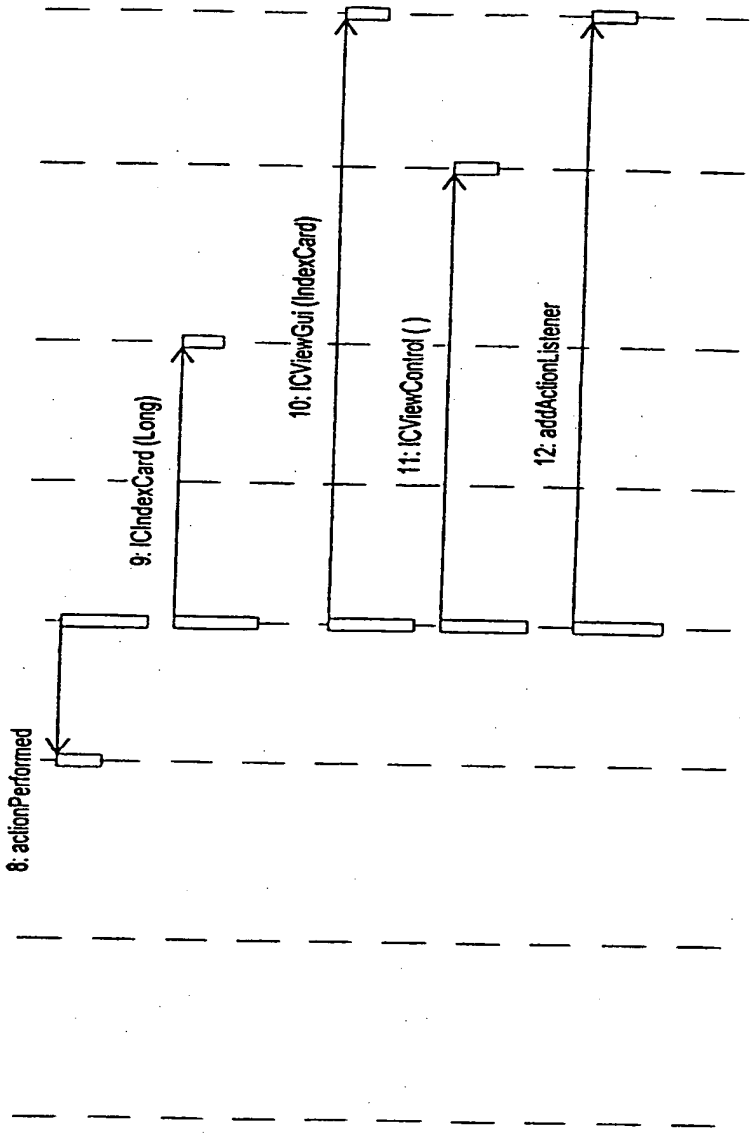

8: actionPerformed
9: ICIndexCard (Long)
10: ICViewGui (IndexCard)
11: ICViewControl ( )
12: addActionListener

Fig. 26b

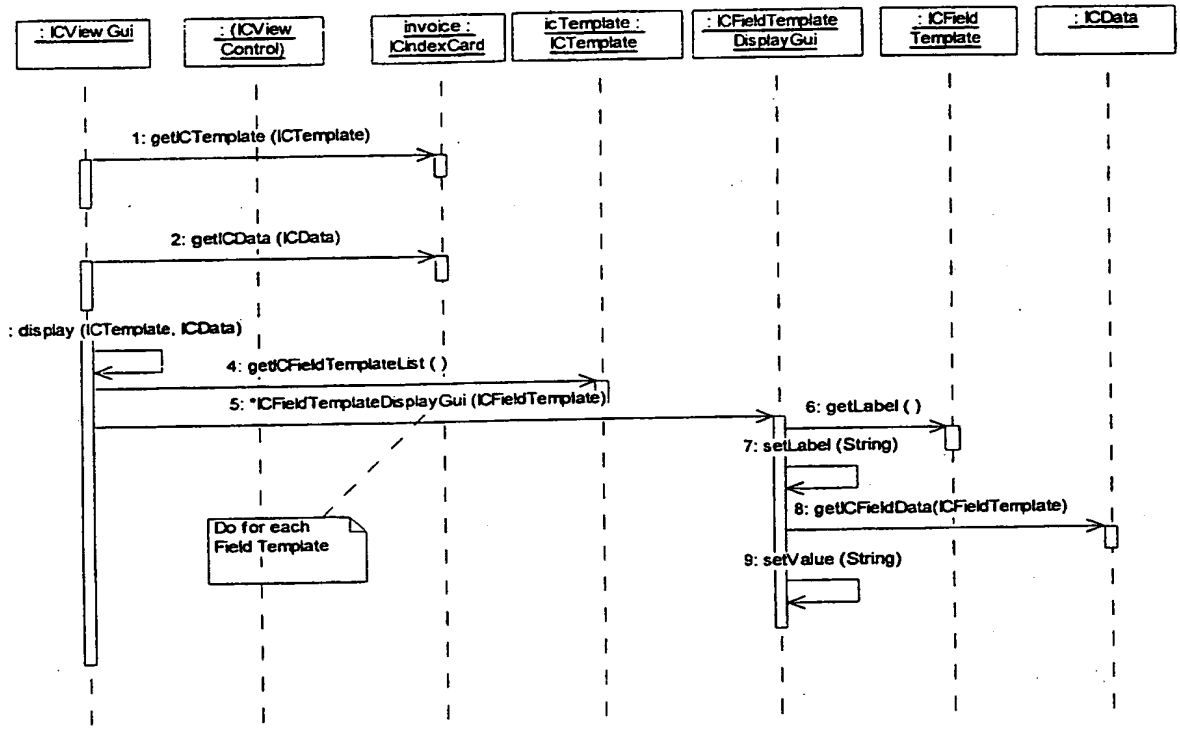

: ICView Gui
: (ICView Control)
invoice : ICIndexCard
icTemplate : ICTemplate
: ICFieldTemplate DisplayGui
: ICField Template
: ICData
1: getICTemplate (ICTemplate)
2: getICData (ICData)
: display (ICTemplate, ICData)
4: getICFieldTemplateList ( )
5: *ICFieldTemplateDisplayGui (ICFieldTemplate)
6: getLabel ( )
7: setLabel (String)
8: getICFieldData(ICFieldTemplate)
9: setValue (String)
Do for each Field Template

Fig. 27

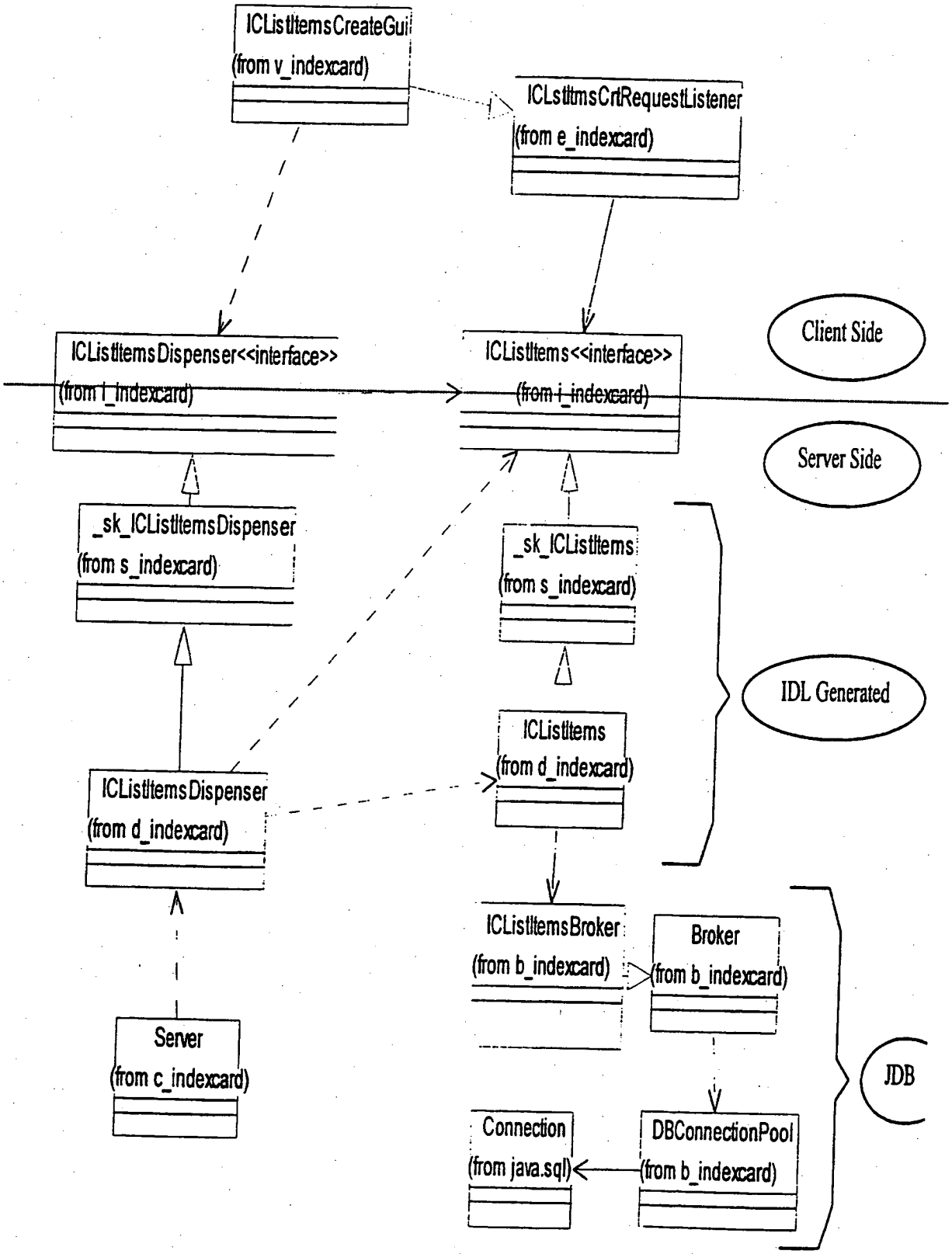

ICListItemsCreateGui
(from v_indexcard)
ICLstItmsCrtRequestListener
(from e_indexcard)
ICListItemsDispenser<<interface>>
(from I_indexcard)
ICListItems<<interface>>
(from i_indexcard)
Client Side
Server Side
_sk_ICListItemsDispenser
(from s_indexcard)
_sk_ICListItems
(from s_indexcard)
IDL Generated
ICListItems
(from d_indexcard)
ICListItemsDispenser
(from d_indexcard)
ICListItemsBroker
(from b_indexcard)
Broker
(from b_indexcard)
Server
(from c_indexcard)
JDB
Connection
(from java.sql)
DBConnectionPool
(from b_indexcard)

Fig. 28

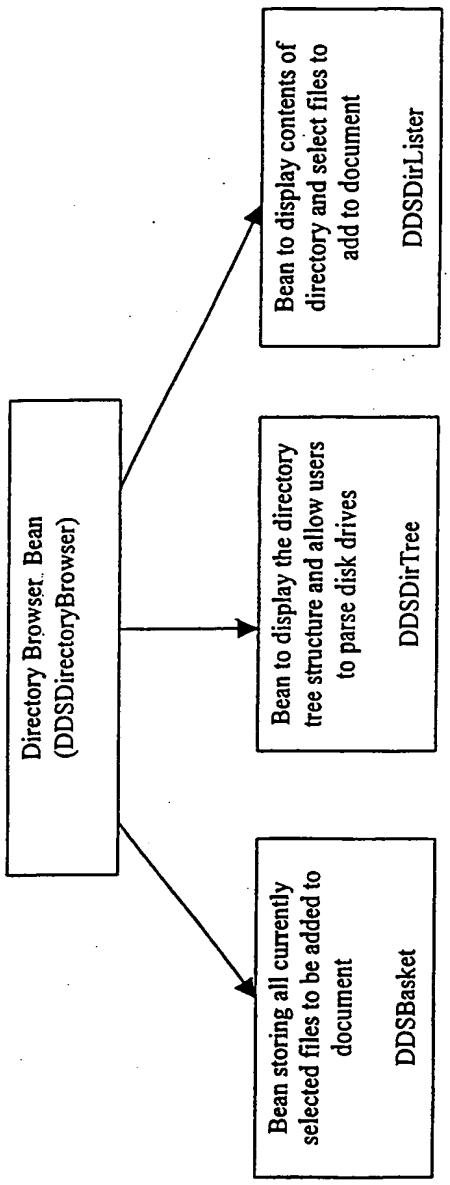

Directory Browser. Bean
(DDSDirectoryBrowser)
Bean to display contents of directory and select files to add to document
DDSDirLister
Bean to display the directory tree structure and allow users to parse disk drives
DDSDirTree
Bean storing all currently selected files to be added to document
DDSBasket

Fig. 29

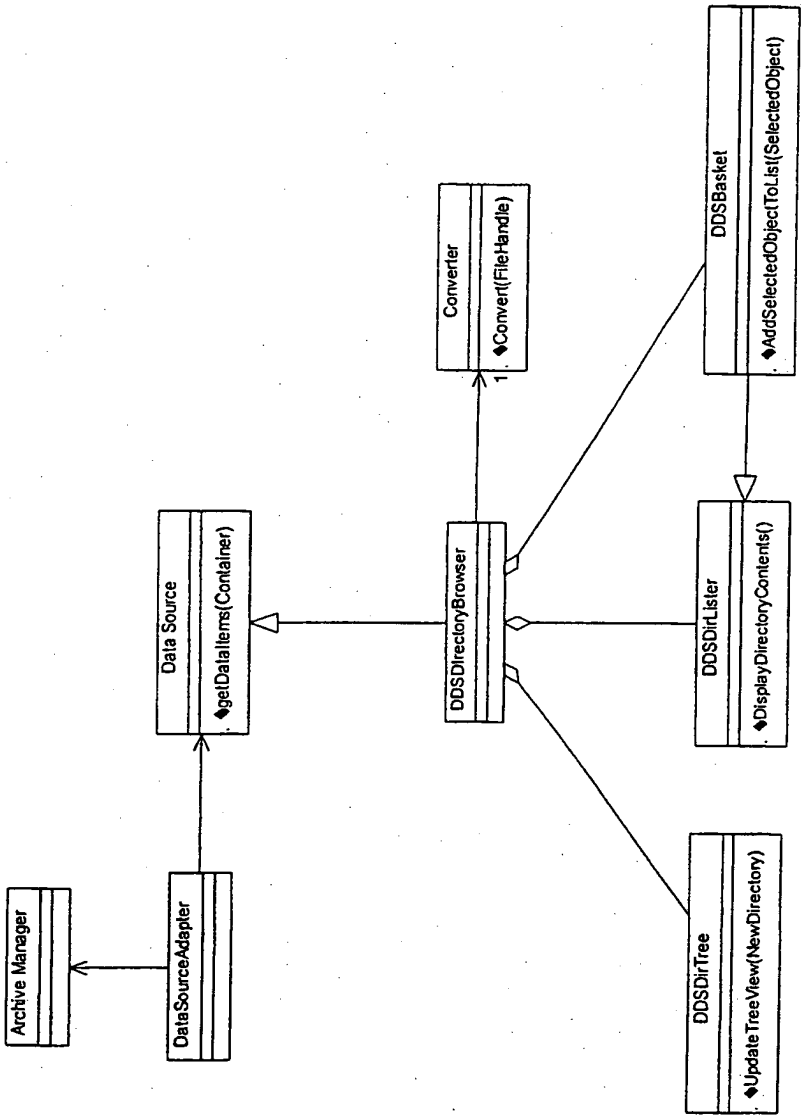

Converter
Convert(FileHandle)
1
DDSBasket
AddSelectedObjectToList(SelectedObject)
Data Source
getDataItems(Container)
DDSDirectoryBrowser
DDSDirLister
DisplayDirectoryContents()
Archive Manager
DataSourceAdapter
DDSDirTree
UpdateTreeView(NewDirectory)

Fig. 30

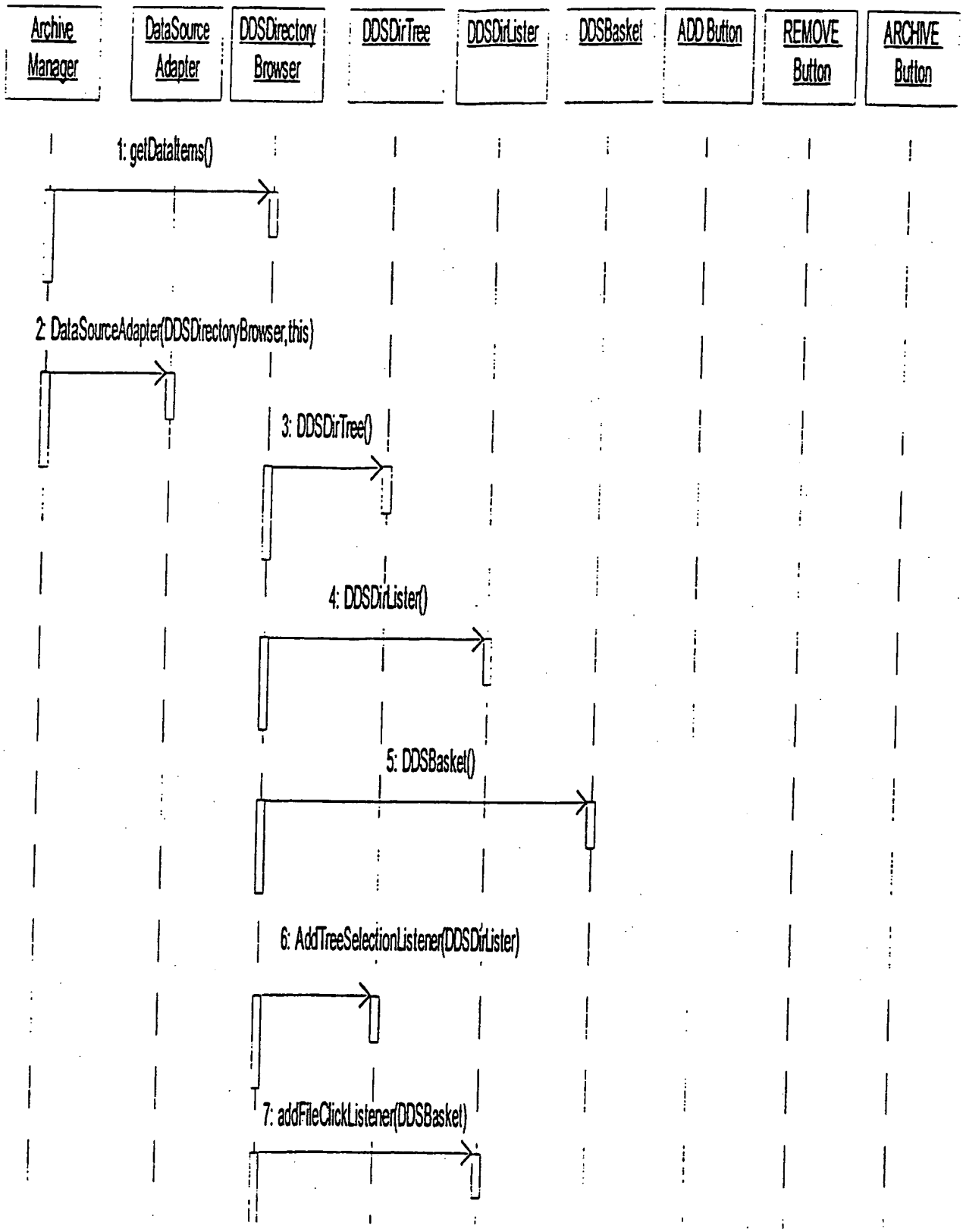

Archive Manager
DataSource Adapter
DDSDirectory Browser
DDSDirTree
DDSDirLister
DDSBasket
ADD Button
REMOVE Button
ARCHIVE Button
1: getDataItems()
2: DataSourceAdapter(DDSDirectoryBrowser,this)
3: DDSDirTree()
4: DDSDirLister()
5: DDSBasket()
6: AddTreeSelectionListener(DDSDirLister)
7: addFileClickListener(DDSBasket)

Fig. 31a

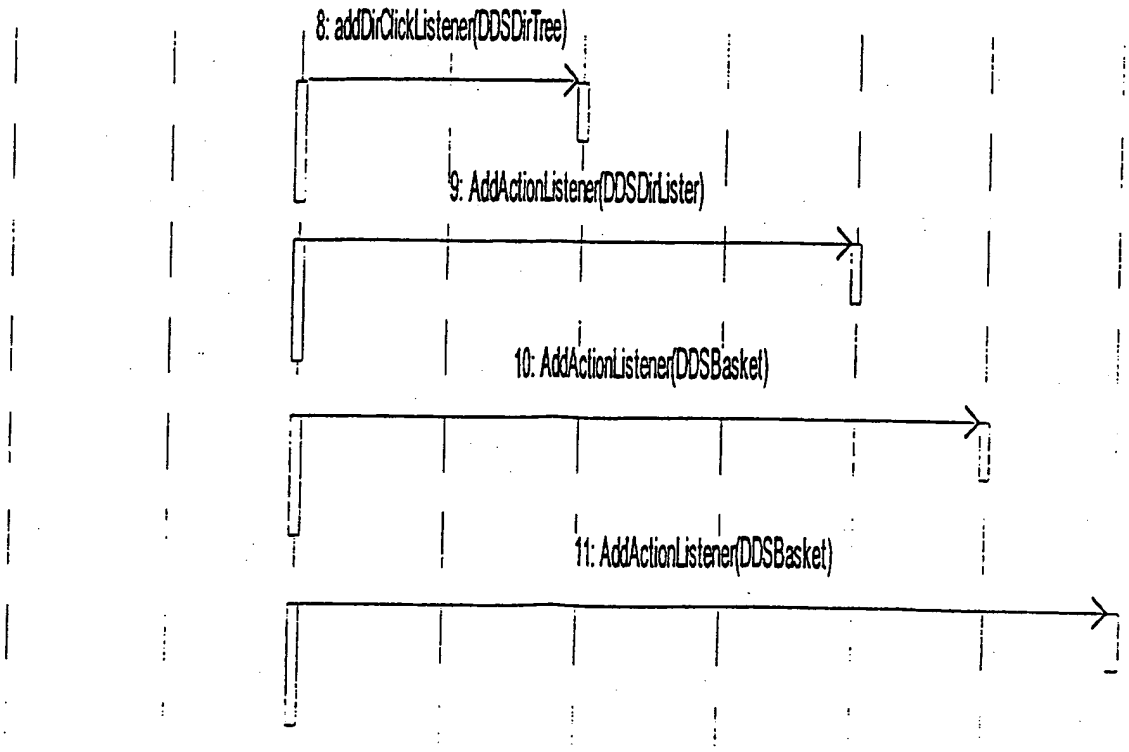

8: addDirClickListener(DDSDirTree)
9: AddActionListener(DDSDirLister)
10: AddActionListener(DDSBasket)
11: AddActionListener(DDSBasket)

Fig. 31b

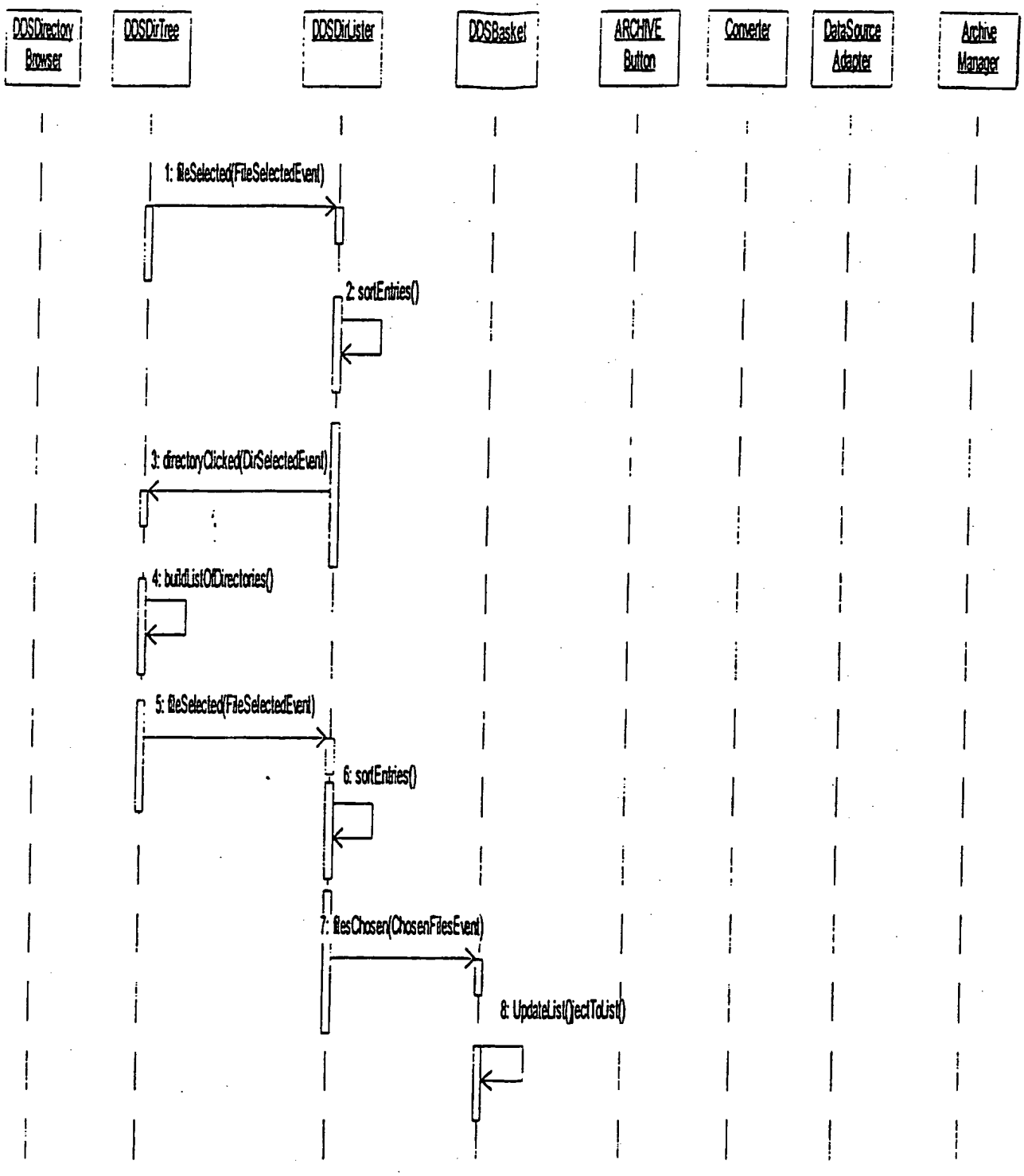

DOSDirectory Browser
DOSDirTree
DOSDirLister
DOSBasket
ARCHIVE Button
Converter
DataSource Adapter
Archive Manager
1: fileSelected(FileSelectedEvent)
2: sortEntries()
3: directoryClicked(DirSelectedEvent)
4: buildListOfDirectories()
5: fileSelected(FileSelectedEvent)
6: sortEntries()
7: filesChosen(ChosenFilesEvent)
8: UpdateList(ObjectToList)

Fig. 32a

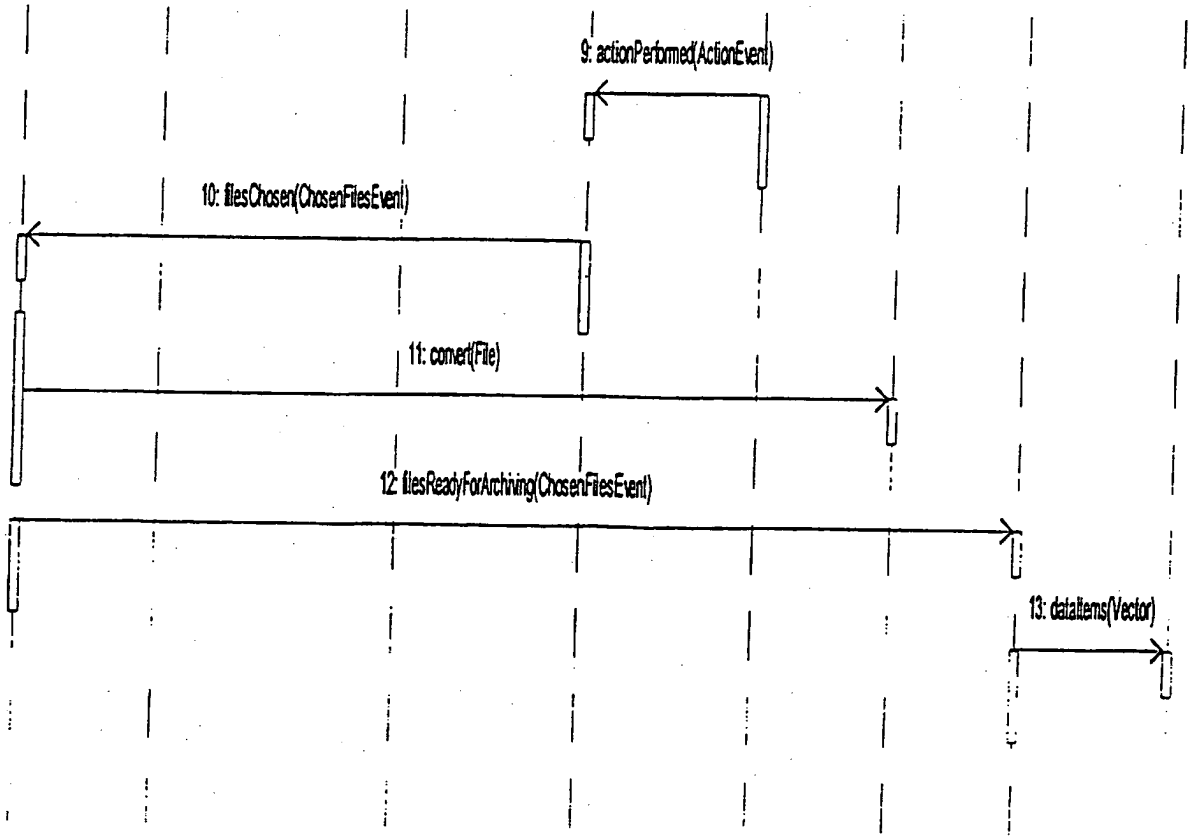

9: actionPerformed(ActionEvent)
10: filesChosen(ChosenFilesEvent)
11: convert(File)
12: filesReadyForArchiving(ChosenFilesEvent)
13: dataItems(Vector)

Fig. 32b

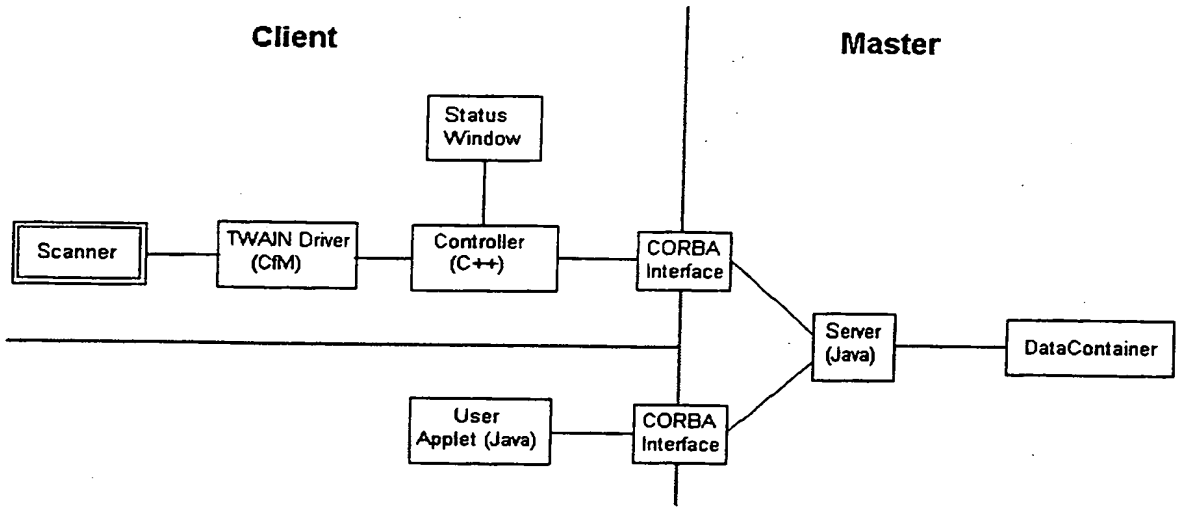

Client
Master
Status Window
Scanner
TWAIN Driver (CfM)
Controller (C++)
CORBA Interface
Server (Java)
DataContainer
User Applet (Java)
CORBA Interface

Fig. 33

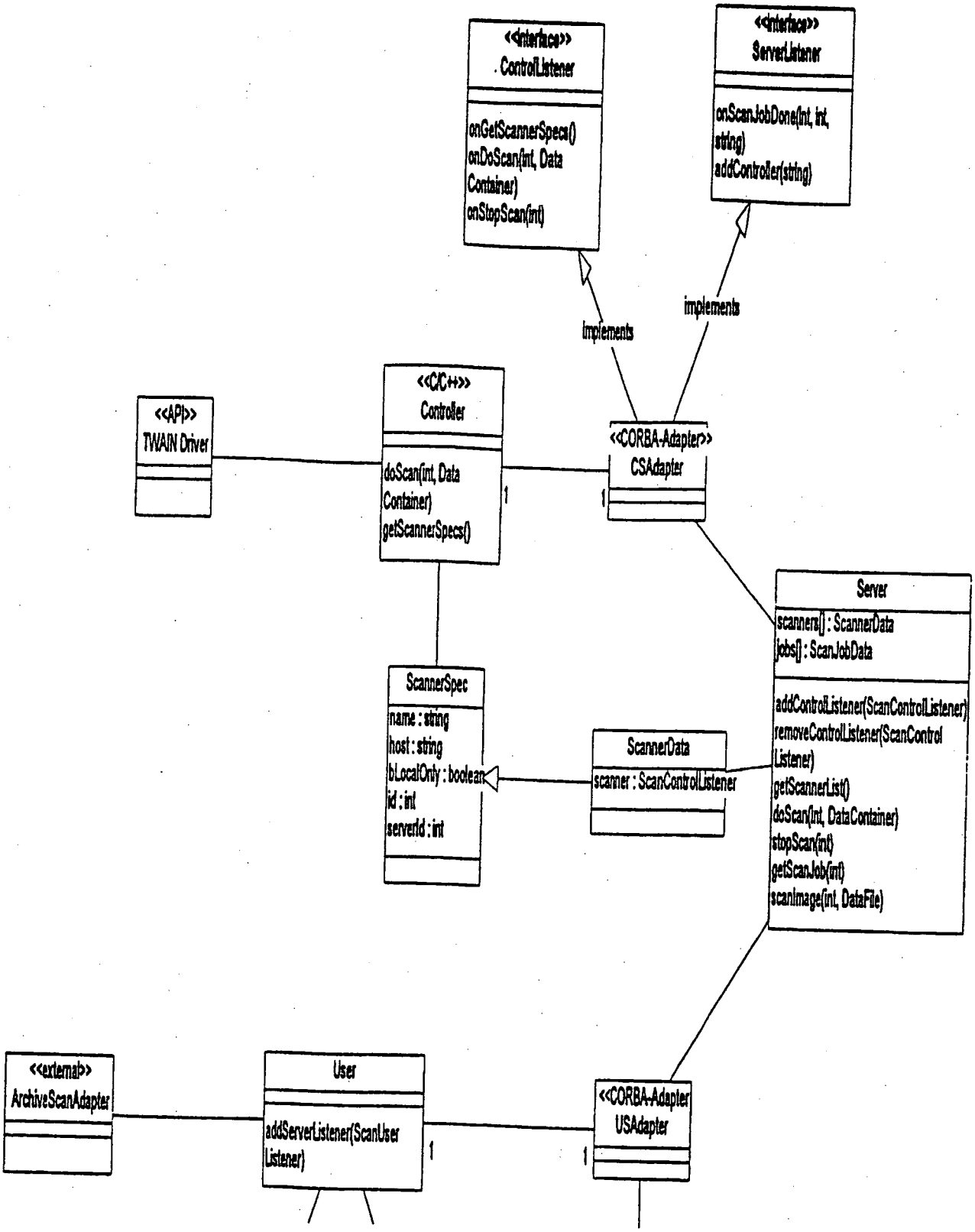

«interface»
ControlListener
onGetScannerSpecs()
onDoScan(int, Data Container)
onStopScan(int)
«interface»
ServerListener
onScanJobDone(int, int, string)
addController(string)
implements
implements
«API»
TWAIN Driver
«C/C++»
Controller
doScan(int, Data Container)
getScannerSpecs()
«CORBA-Adapter»
CSAdapter
1
1
Server
scanners[] : ScannerData
jobs[] : ScanJobData
addControlListener(ScanControlListener)
removeControlListener(ScanControl Listener)
getScannerList()
doScan(int, DataContainer)
stopScan(int)
getScanJob(int)
scanImage(int, DataFile)
ScannerSpec
name : string
host : string
bLocalOnly : boolean
id : int
serverId : int
ScannerData
scanner : ScanControlListener
«external»
ArchiveScanAdapter
User
addServerListener(ScanUser Listener)
1
1
«CORBA-Adapter»
USAdapter

Fig. 34a

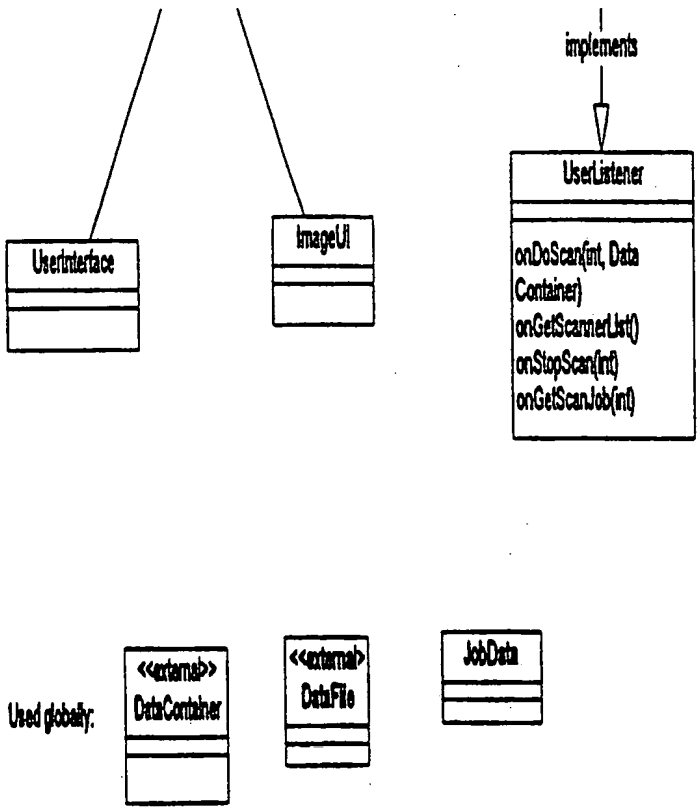

implements
UserInterface
ImageUI
UserListener
onDoScan(int, Data Container)
onGetScannerList()
onStopScan(int)
onGetScanJob(int)
Used globally:
<<external>> DataContainer
<<external>> DataFile
JobData

Fig. 34b

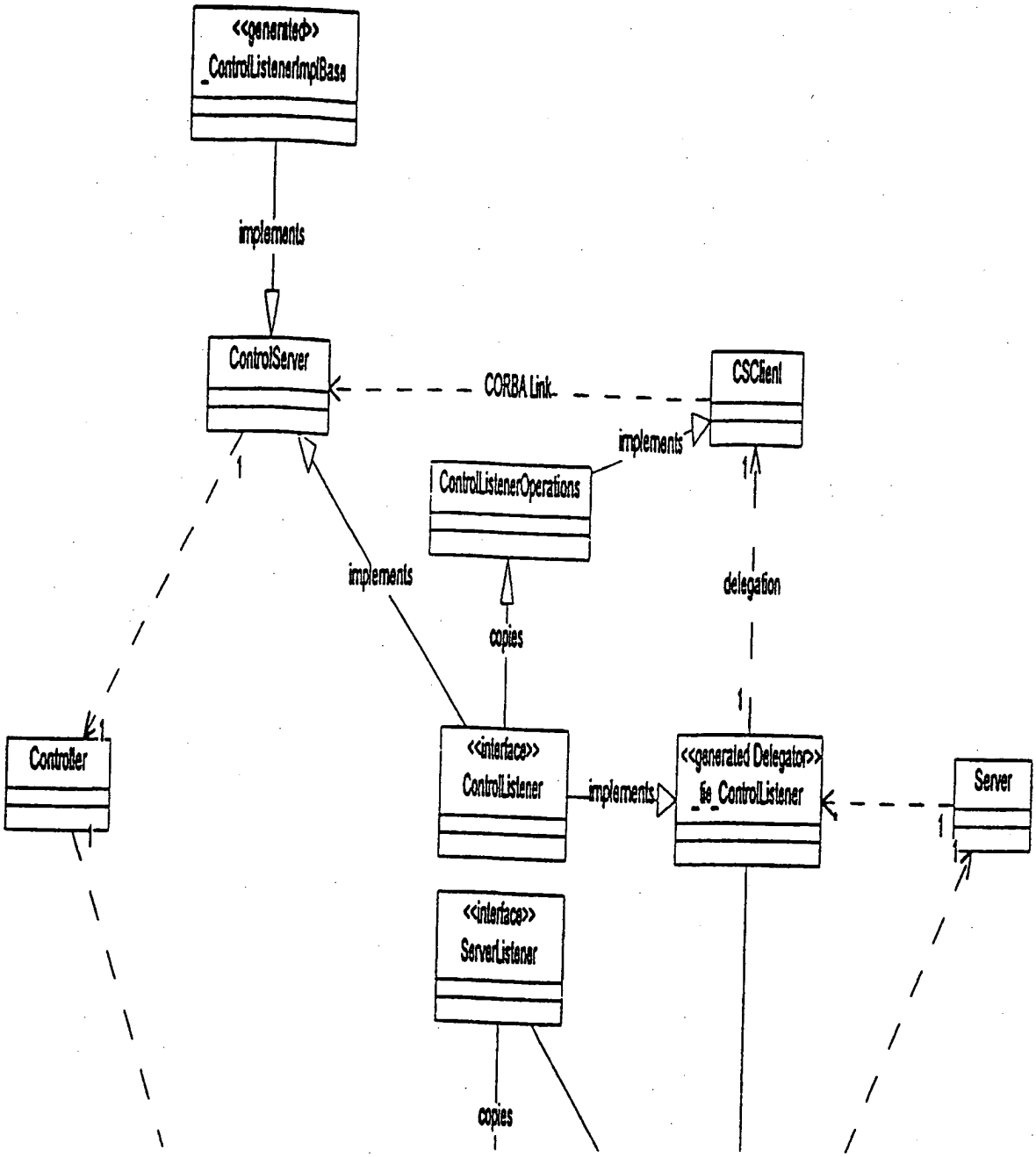

«generated»
_ControlListenerImplBase
implements
ControlServer
CORBA Link
CSClient
implements
ControlListenerOperations
1
1
implements
delegation
copies
1
1
Controller
«interface»
ControlListener
implements
«generated Delegator»
_tie_ControlListener
Server
1
«interface»
ServerListener
copies

Fig. 35a

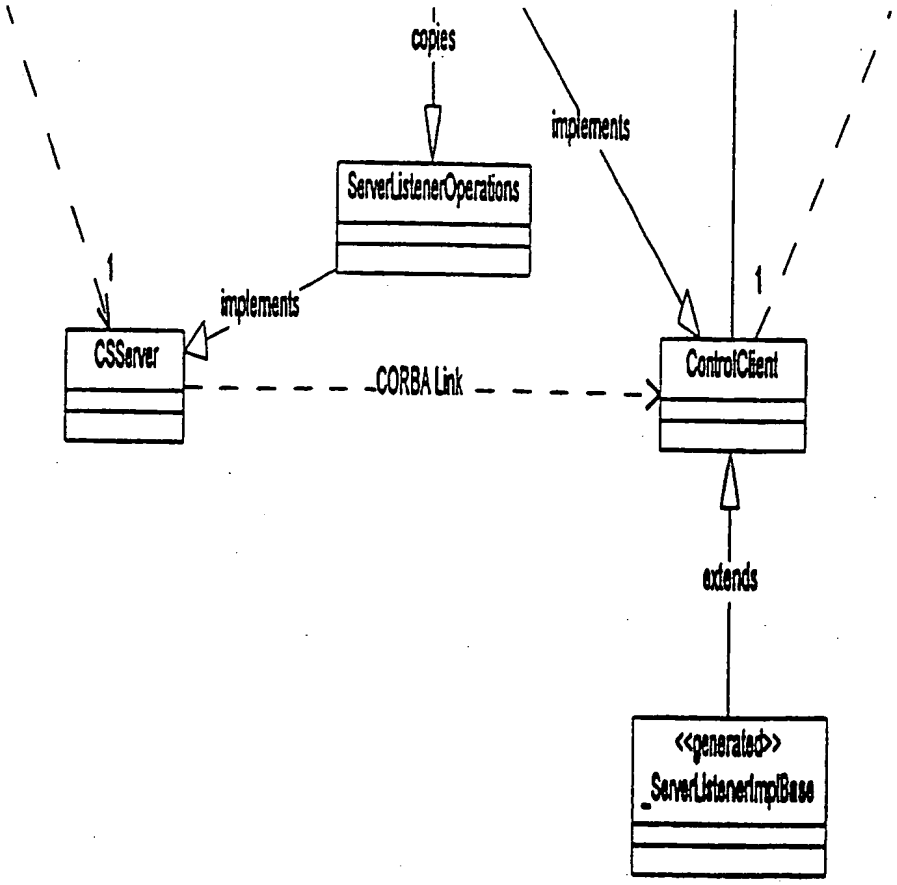

copies
implements
ServerListenerOperations
1
implements
1
CSServer
CORBA Link
ControlClient
extends
«generated»
_ServerListenerImplBase

Fig. 35b

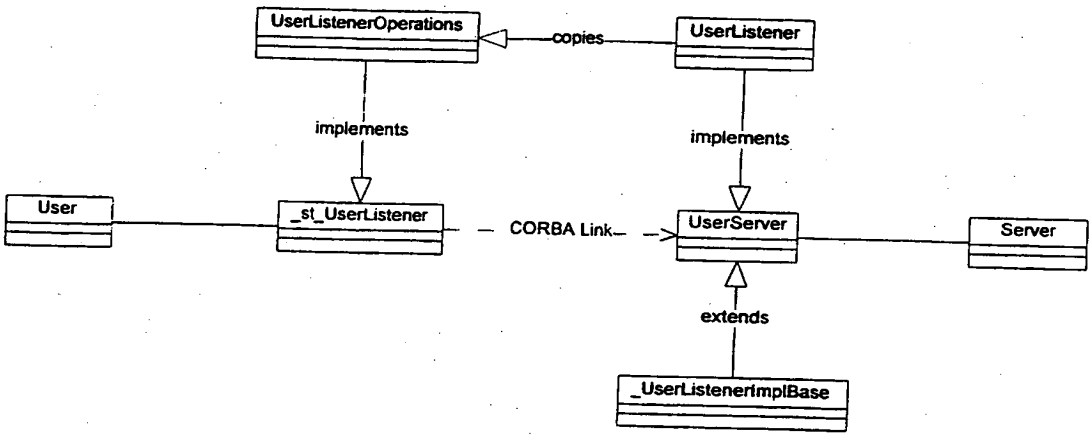

UserListenerOperations
copies
UserListener
implements
implements
User
_st_UserListener
CORBA Link
UserServer
Server
extends
_UserListenerImplBase

Fig. 36

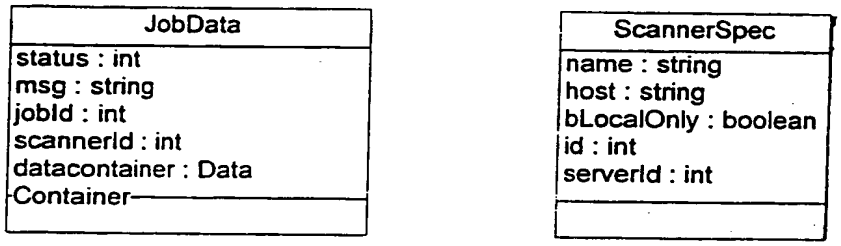

JobData
status : int
msg : string
jobId : int
scannerId : int
datacontainer : Data
Container
ScannerSpec
name : string
host : string
bLocalOnly : boolean
id : int
serverId : int

Fig. 37

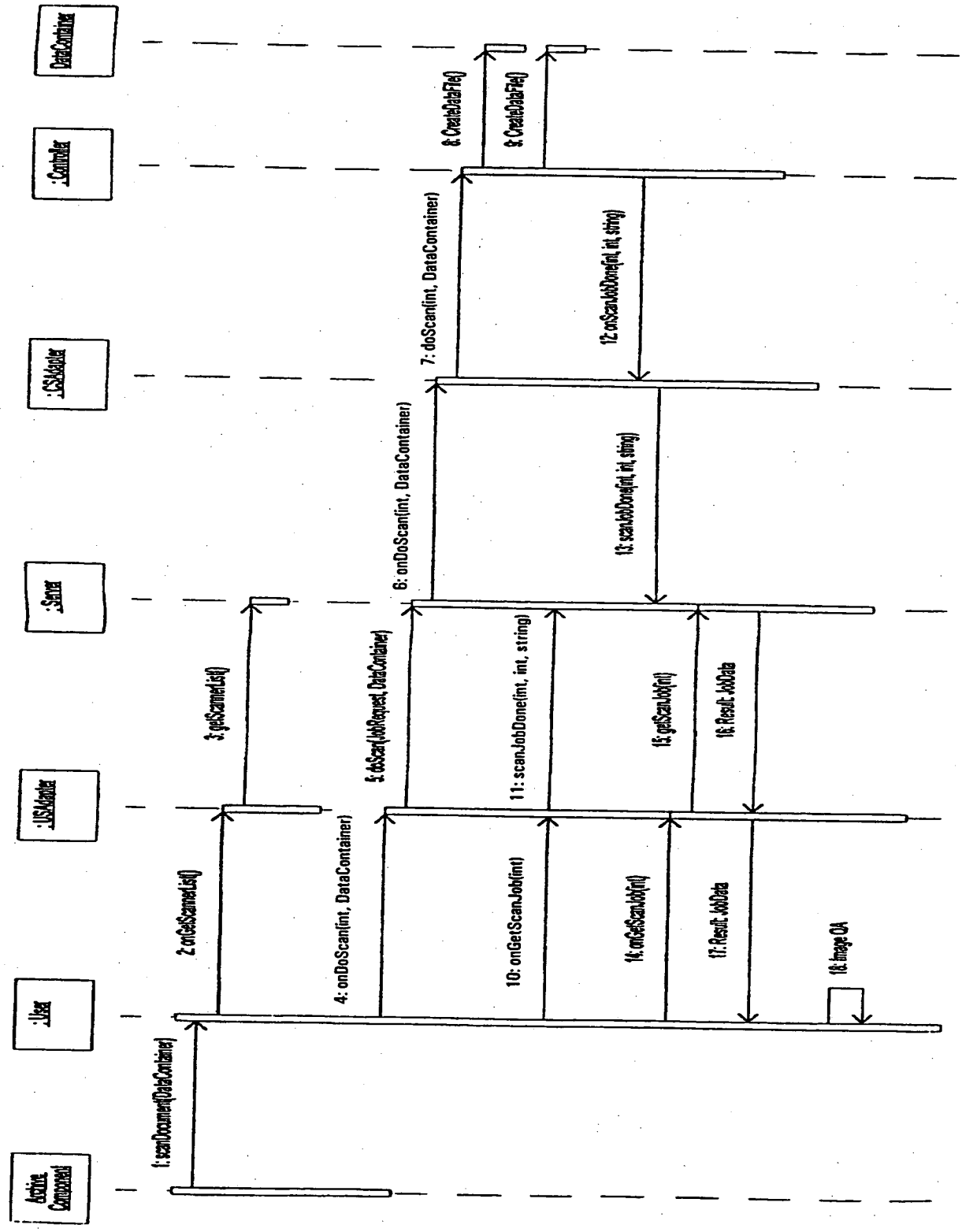

Active Component
:User
:IScMediator
:Server
:IScMediator
:Controller
DataContainer
1: scanDocument(DataContainer)
2: onGetScannerList()
3: getScannerList()
4: onDoScan(int, DataContainer)
5: doScan(JobRequest, DataContainer)
6: onDoScan(int, DataContainer)
7: doScan(int, DataContainer)
8: CreateDataFile()
8: CreateDataFile()
10: onGetScanJob(int)
11: scanJobDone(int, int, string)
12: onScanJobDone(int, int, string)
13: scanJobDone(int, int, string)
14: onGetScanJob(int)
15: getScanJob(int)
16: Result: JobData
17: Result: JobData
18: Image OA

Fig. 38

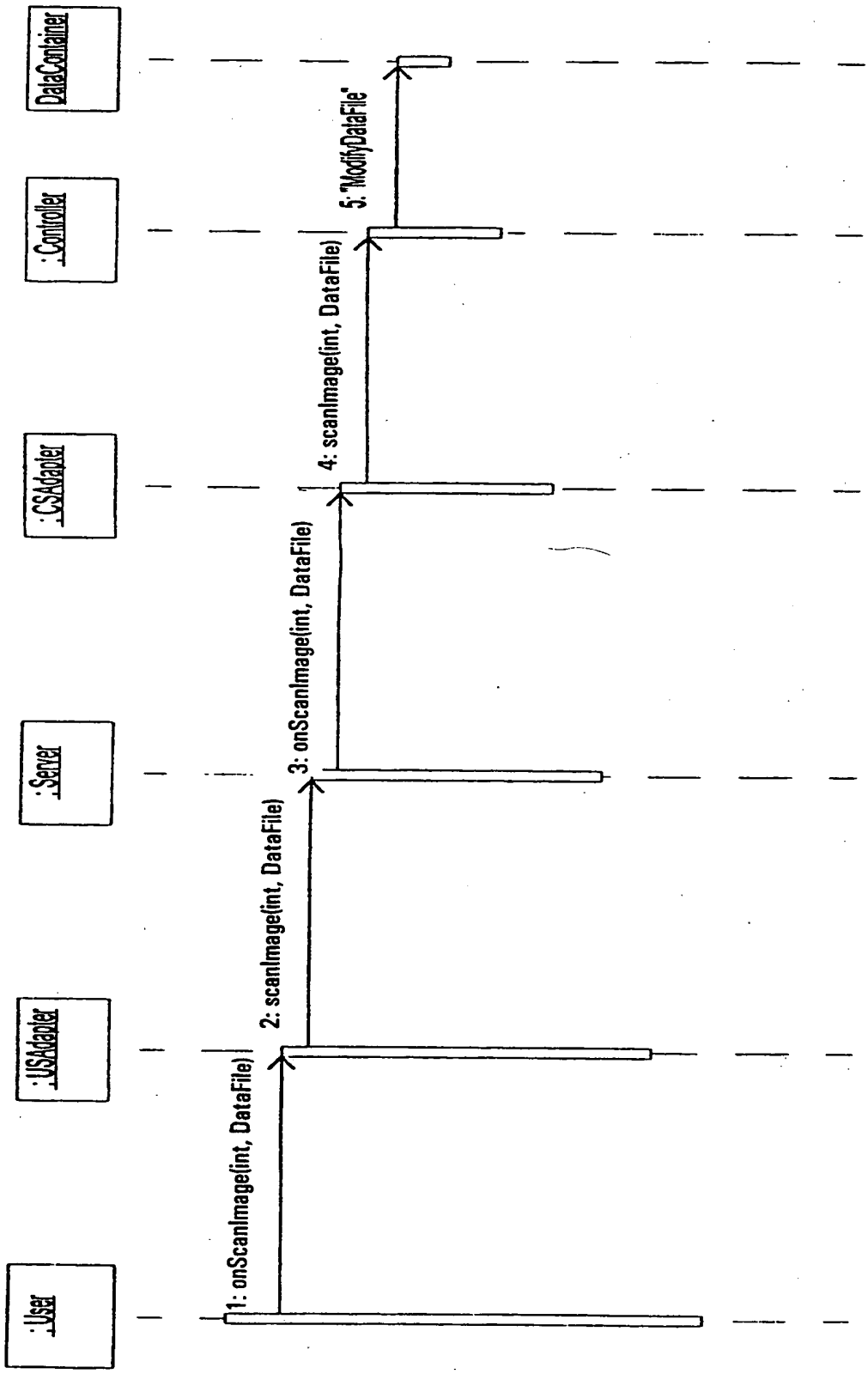

:User
:USAdapter
:Server
:CSAdapter
:Controller
DataContainer
1: onScanImage(int, DataFile)
2: scanImage(int, DataFile)
3: onScanImage(int, DataFile)
4: scanImage(int, DataFile)
5: 'ModifyDataFile'

Fig. 39

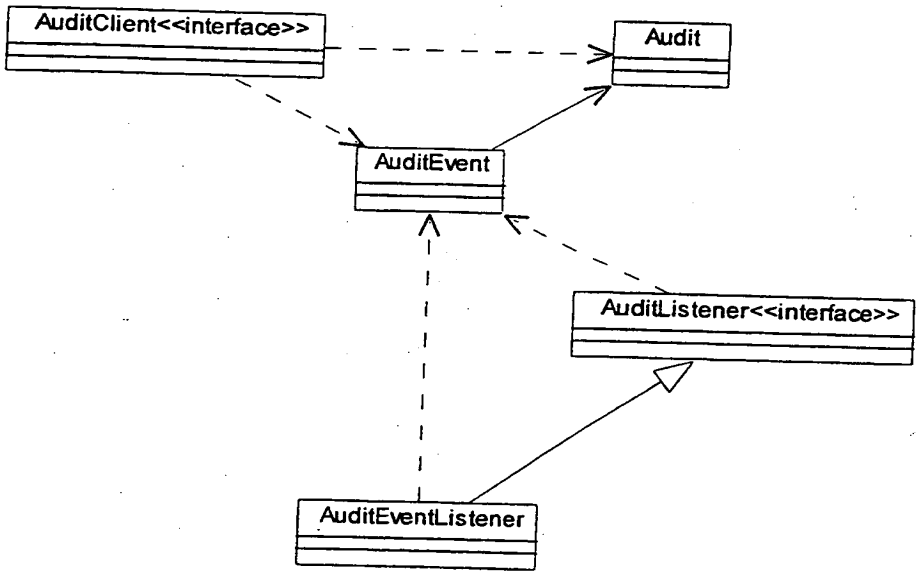

AuditClient<<interface>>
Audit
AuditEvent
AuditListener<<interface>>
AuditEventListener

Fig. 40

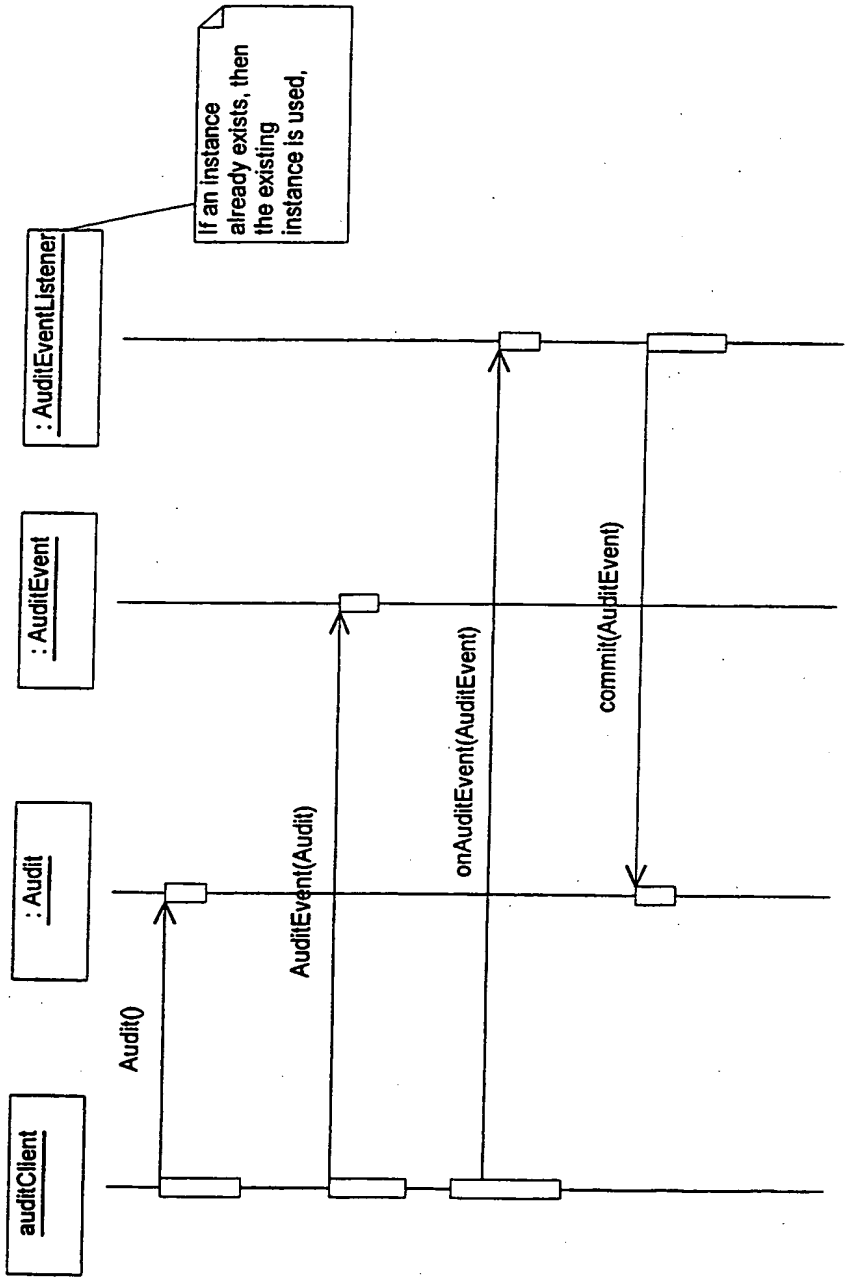

If an instance already exists, then the existing instance is used,
: AuditEventListener
: AuditEvent
: Audit
auditClient
Audit()
AuditEvent(Audit)
onAuditEvent(AuditEvent)
commit(AuditEvent)

Fig. 41

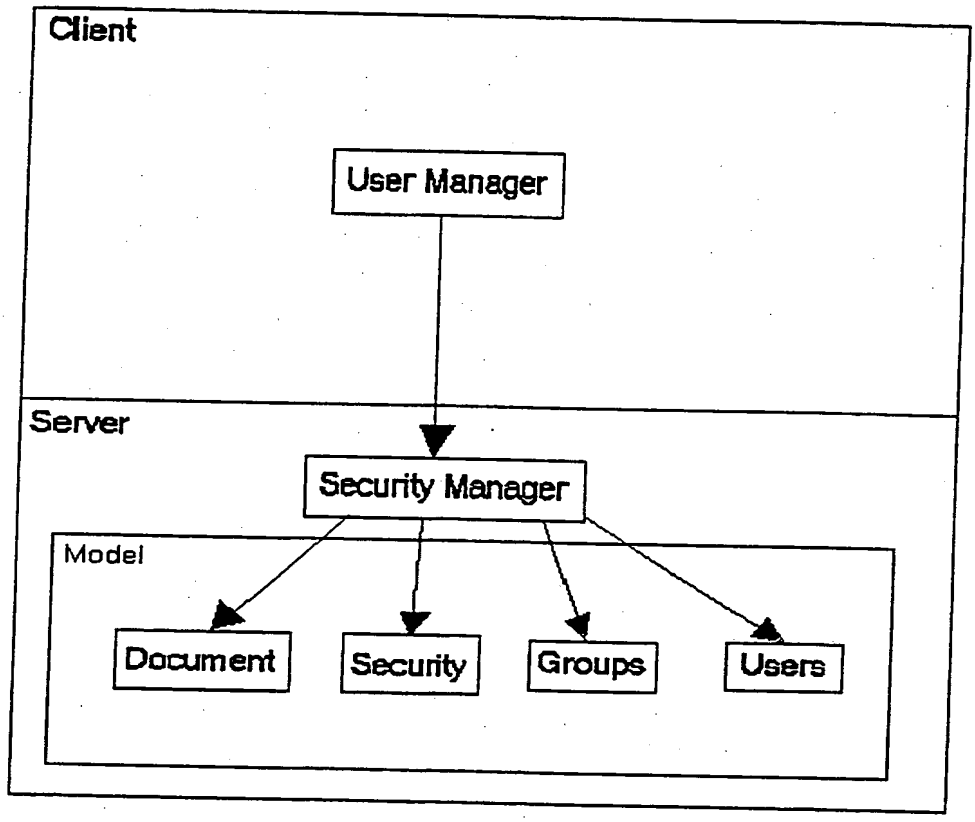

Client
User Manager
Server
Security Manager
Model
Document
Security
Groups
Users

Fig. 42

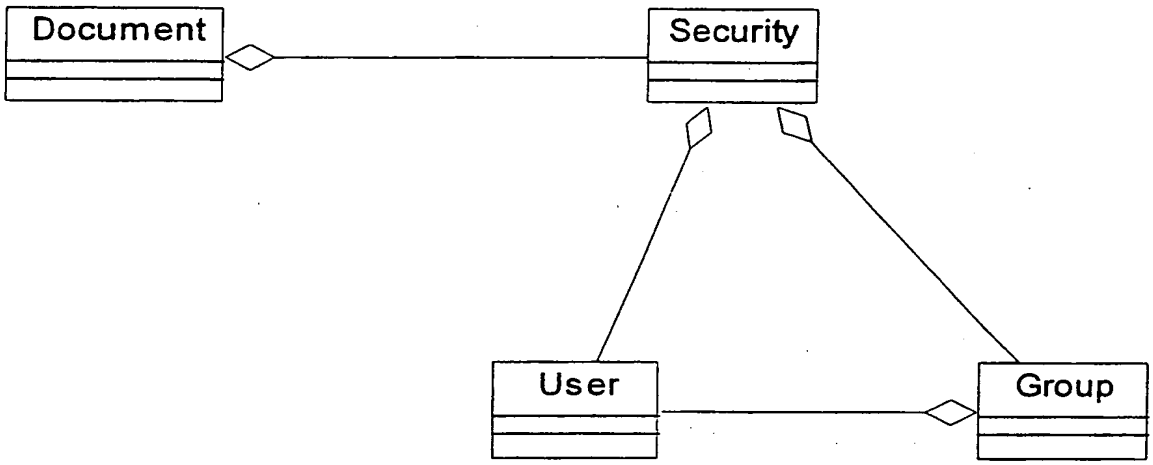


Fig. 43

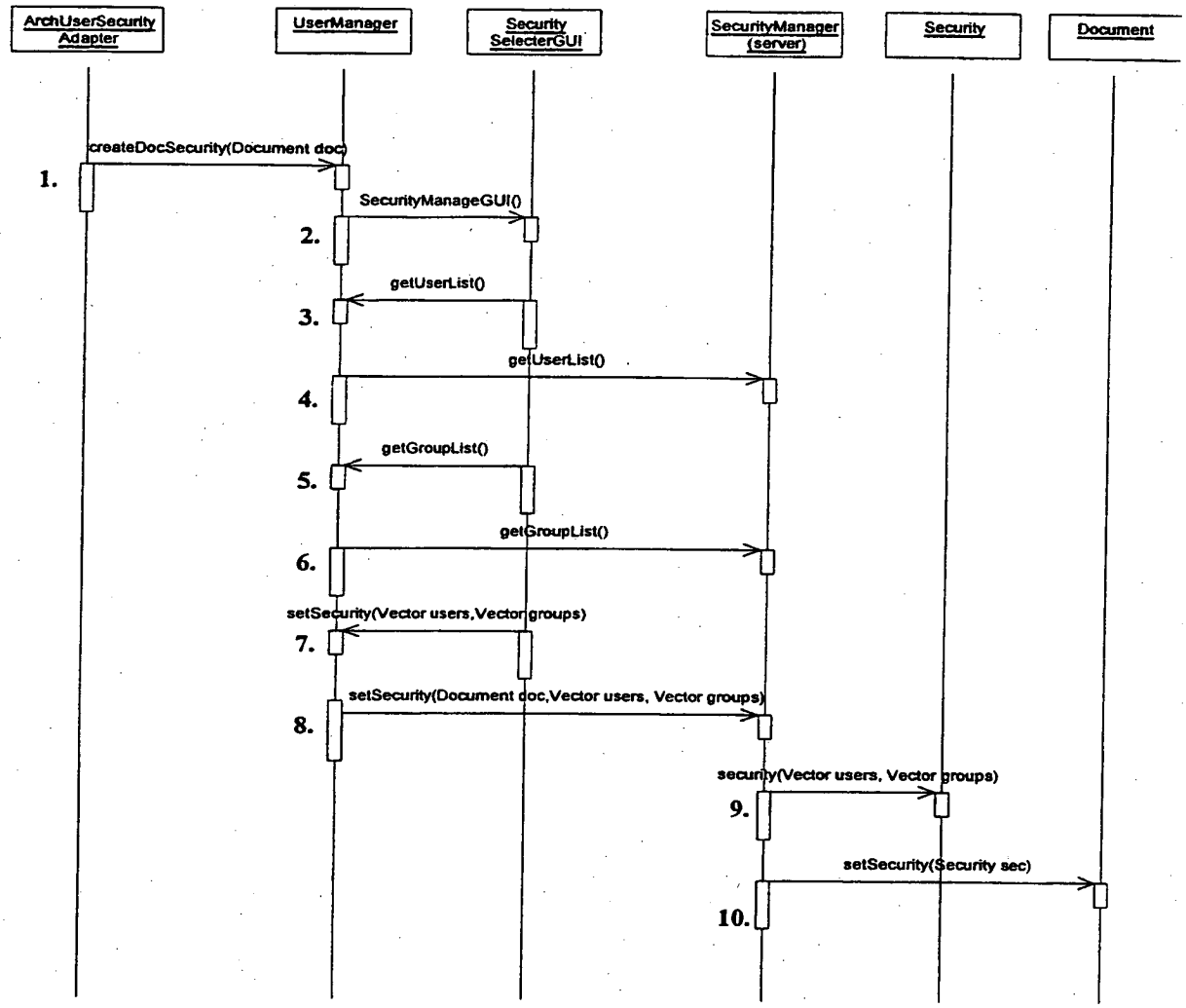

ArchUserSecurity Adapter
UserManager
Security SelecterGUI
SecurityManager (server)
Security
Document
createDocSecurity(Document doc)
1.
SecurityManageGUI()
2.
getUserList()
3.
getUserList()
4.
getGroupList()
5.
getGroupList()
6.
setSecurity(Vector users,Vector groups)
7.
setSecurity(Document doc,Vector users, Vector groups)
8.
security(Vector users, Vector groups)
9.
setSecurity(Security sec)
10.

Fig. 44

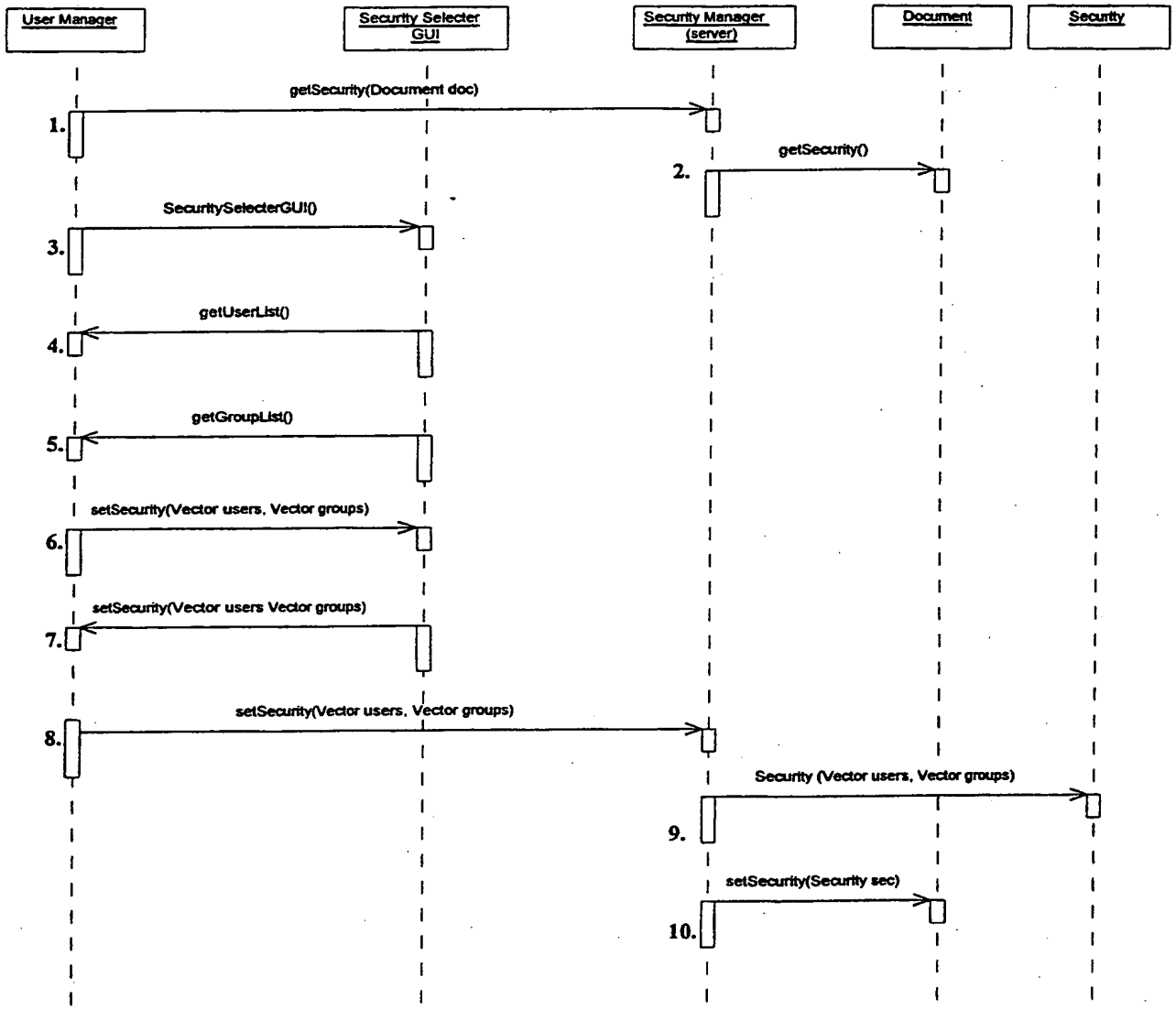

User Manager
Security Selecter GUI
Security Manager (server)
Document
Security
getSecurity(Document doc)
1.
getSecurity()
2.
SecuritySelecterGUI()
3.
getUserList()
4.
getGroupList()
5.
setSecurity(Vector users, Vector groups)
6.
setSecurity(Vector users Vector groups)
7.
setSecurity(Vector users, Vector groups)
8.
Security (Vector users, Vector groups)
9.
setSecurity(Security sec)
10.

Fig. 45

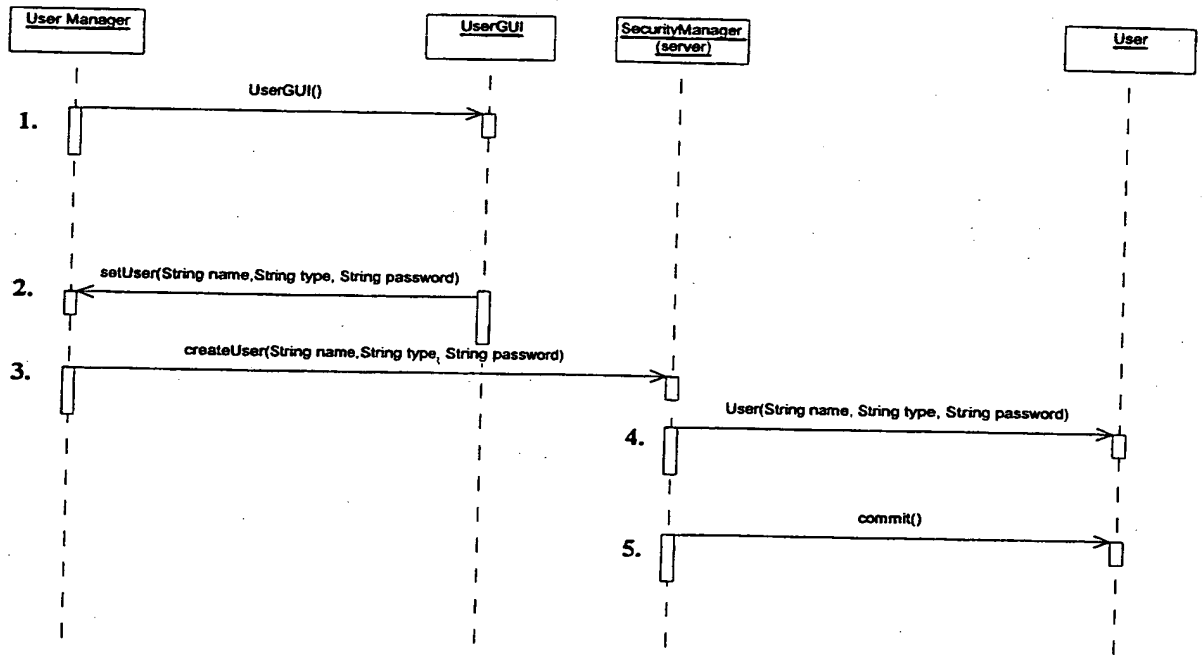

User Manager
UserGUI
SecurityManager (server)
User
1.
UserGUI()
2.
setUser(String name,String type, String password)
3.
createUser(String name,String type, String password)
4.
User(String name, String type, String password)
5.
commit()

Fig. 46

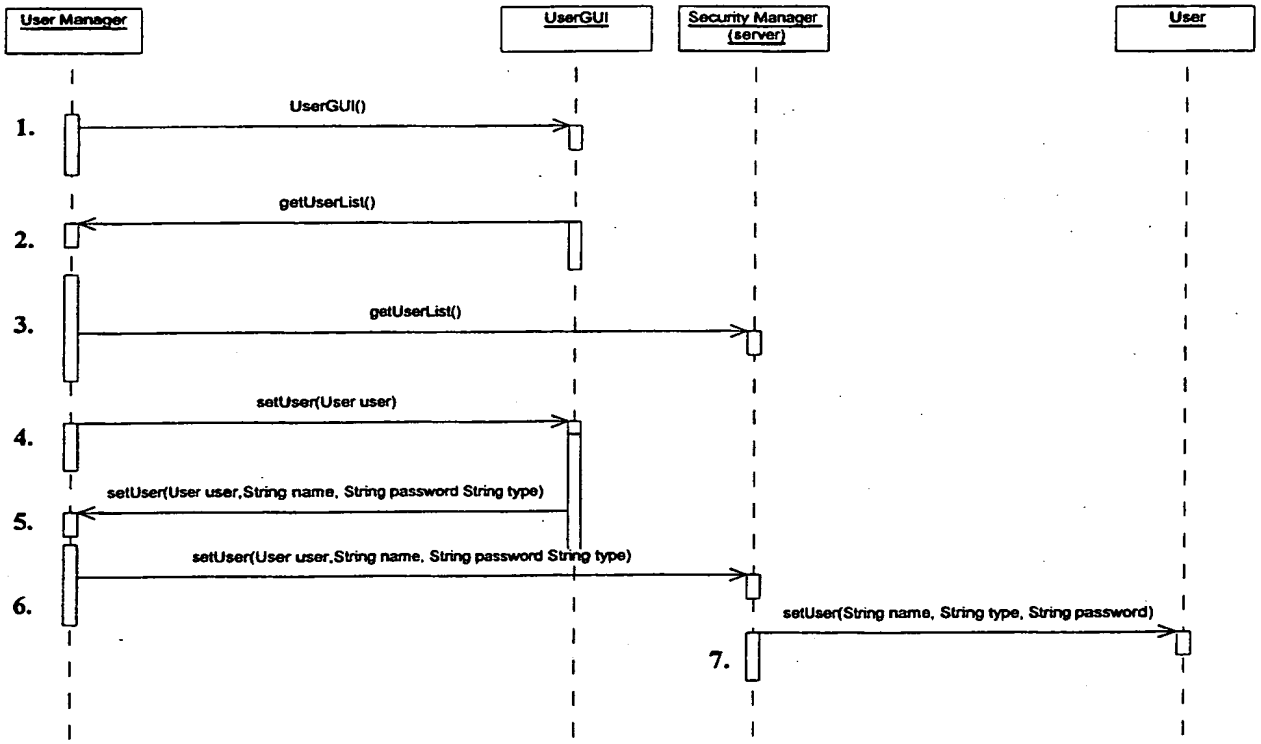

User Manager
UserGUI
Security Manager (server)
User
1.
UserGUI()
2.
getUserList()
3.
getUserList()
4.
setUser(User user)
5.
setUser(User user,String name, String password String type)
6.
setUser(User user,String name, String password String type)
7.
setUser(String name, String type, String password)

Fig. 47

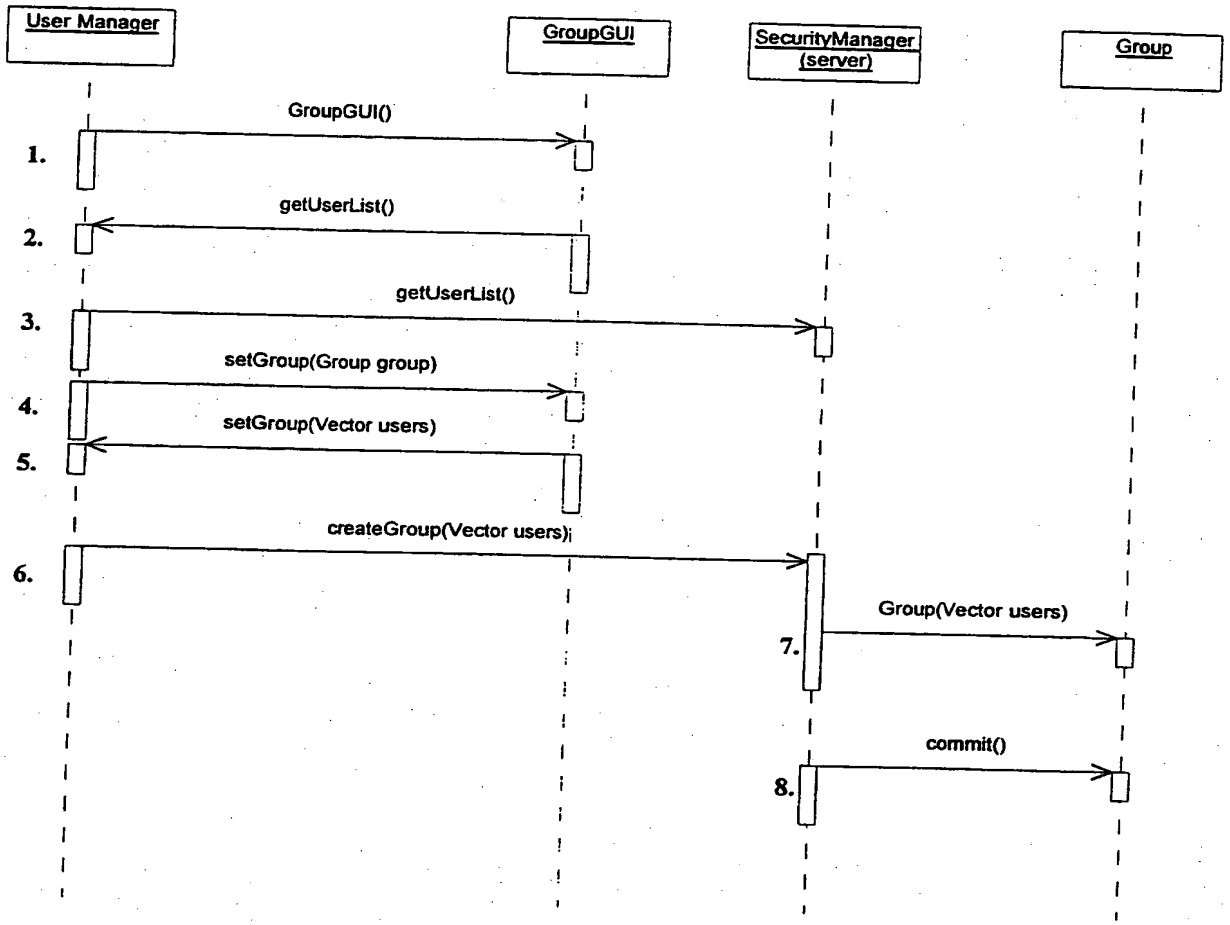

User Manager
GroupGUI
SecurityManager (server)
Group
GroupGUI()
1.
getUserList()
2.
getUserList()
3.
setGroup(Group group)
4.
setGroup(Vector users)
5.
createGroup(Vector users)
6.
Group(Vector users)
7.
commit()
8.

Fig. 48

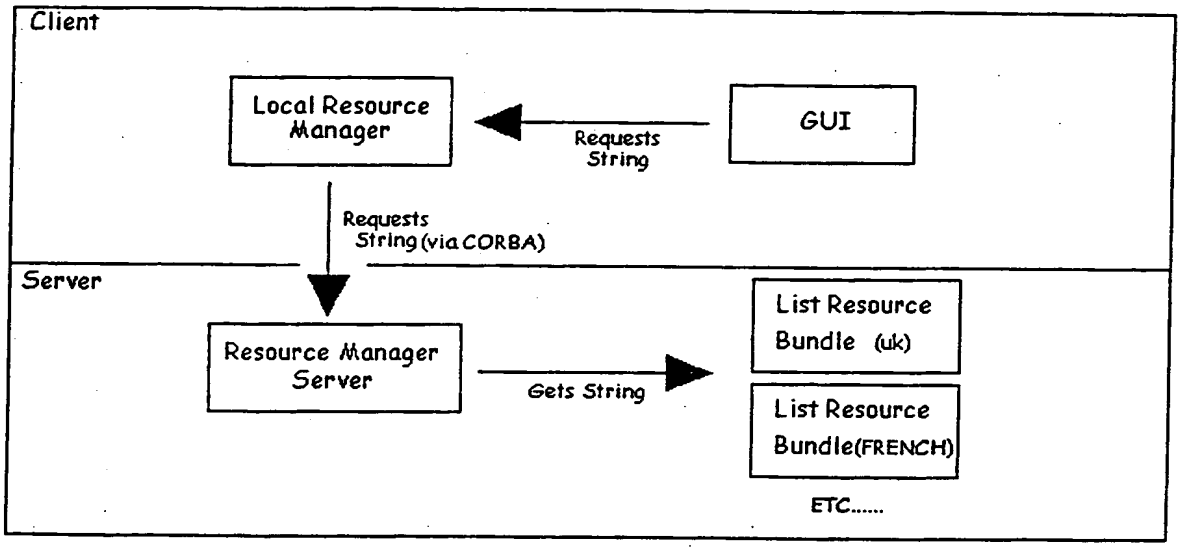

Client
Local Resource Manager
Requests String
GUI
Requests String (via CORBA)
Server
Resource Manager Server
Gets String
List Resource Bundle (uk)
List Resource Bundle(FRENCH)
ETC.....

Fig. 49

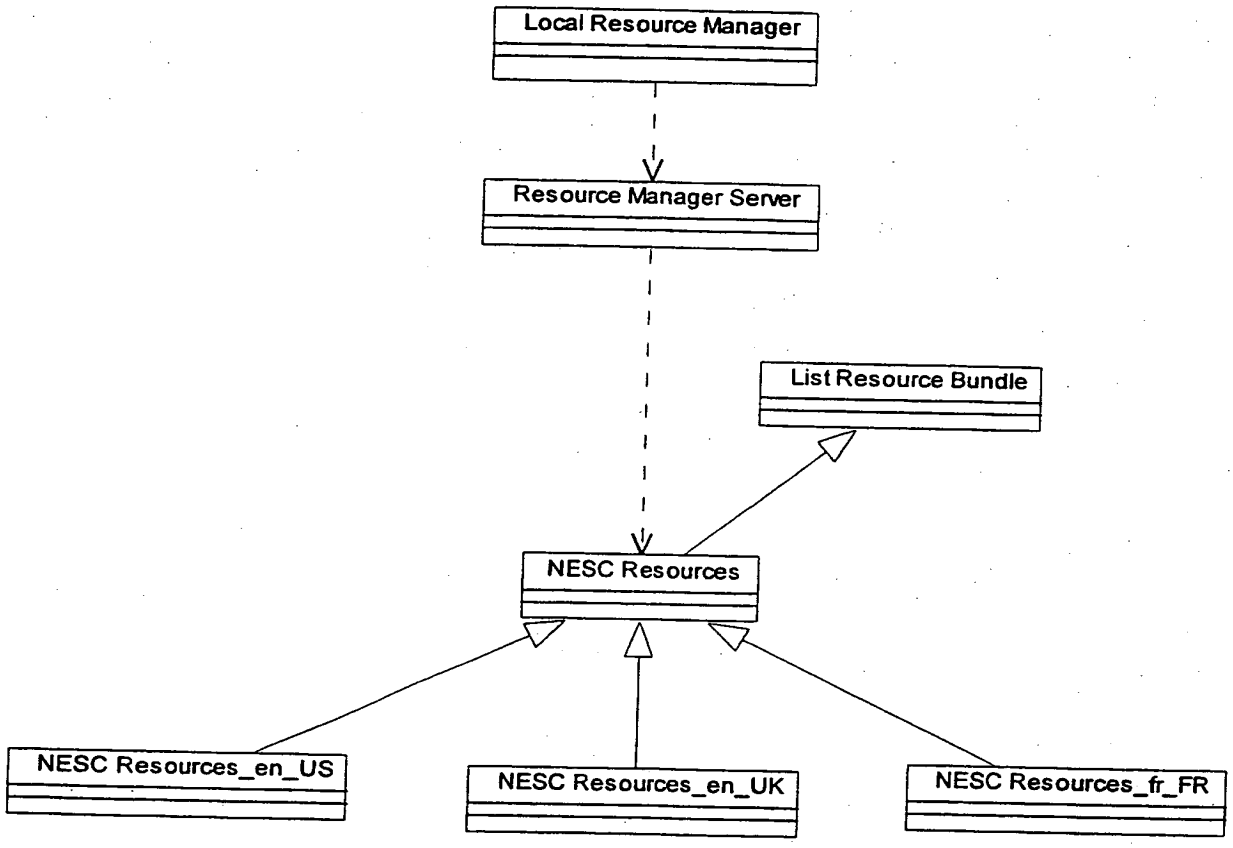

Local Resource Manager
Resource Manager Server
List Resource Bundle
NESC Resources
NESC Resources_en_US
NESC Resources_en_UK
NESC Resources_fr_FR

Fig. 50

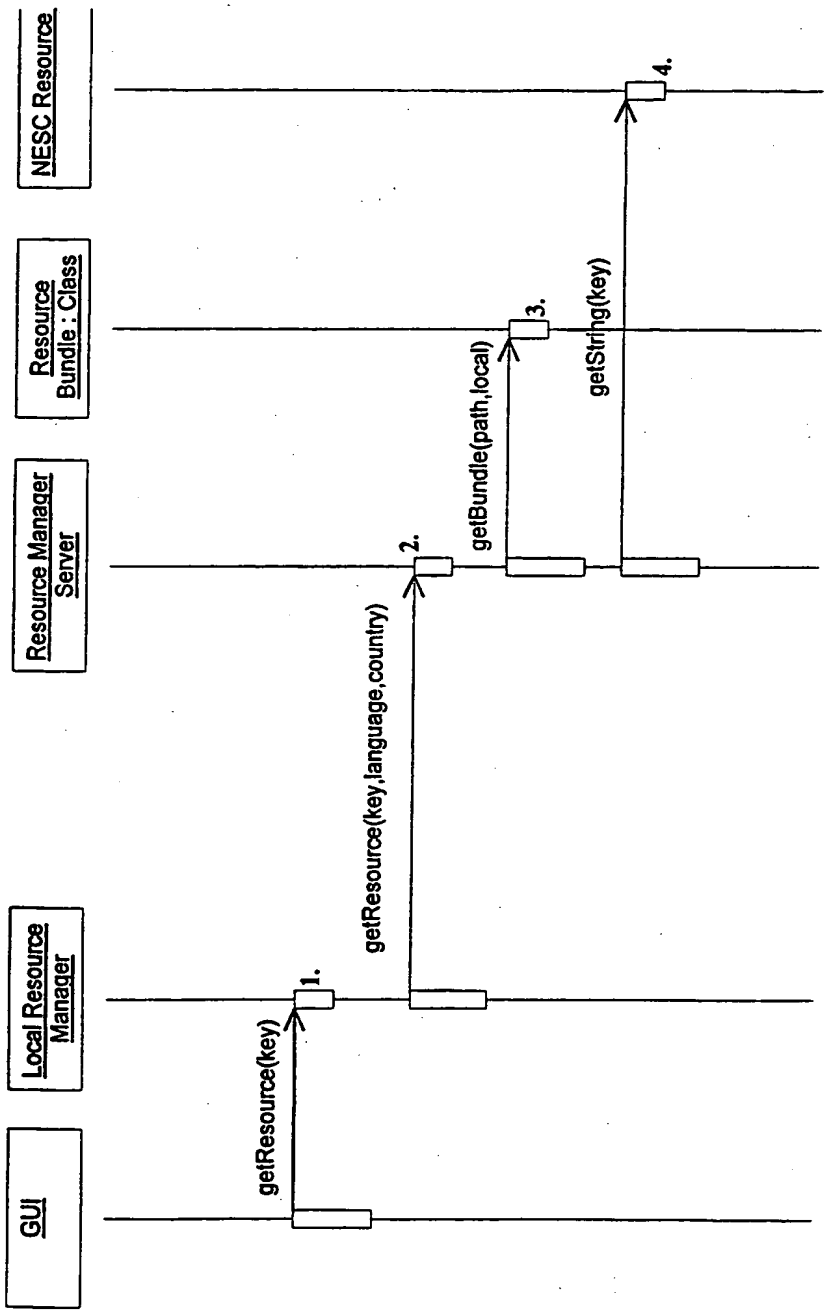

NESC Resource
Resource Bundle : Class
Resource Manager Server
Local Resource Manager
GUI
1.
2.
3.
4.
getResource(key)
getResource(key,language,country)
getBundle(path,local)
getString(key)

Fig. 51

INTERNATIONAL SEARCH REPORT

Inte ional Application No
PCT/IB 99/00878

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F17/30 G06F17/60

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category ³	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	ABERDEEN GROUP, INC.: "Process-centric Applications Drive the Bottom Line: NovaSoft Marries the Web and the Business Process" NOVASOFT SYSTEMS INC. PRESS AREA,1997, XP002090204 htp://www.novasoft.com/web/press/aberdeen.html the whole document ---	1-47
A	MATRIXONE, INC.: "Product Description" MATRIXONE: PRODUCTS - MATRIX,1997, page 1-7 XP002111377 http://www.matrix-one.com/matrixprod.htm the whole document --- -/--	1-47

[X] Further documents are listed in the continuation of box C.

[ ] Patent family members are listed in annex.

³ Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
5 August 1999	17/08/1999

Name and mailing address of the ISA
European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040, Tx. 31 651 epo nl,
Fax: (+31-70) 340-3016

Authorized officer
Abbing, R

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inter onal Application No
PCT/IB 99/00878

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	GRALA A ET AL: "GDOC: a system for storage and authoring of documents through WEB browsers" PROCEEDINGS. 17TH INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY (CAT. NO.97TB100194), PROCEEDINGS 17TH INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY, VALPARAISO, CHILE, 10-15 NOV. 1997, pages 115-124, XP002090205 ISBN 0-8186-8052-0, 1997, Los Alamitos, CA, USA, IEEE Comput. Soc, USA the whole document	1-47

1